

## Menu

## Tech.Pro

- [Content](#)
- [Network](#)
- [More](#)
- [Login](#)
- [Create a Profile](#)

- [Create a Profile](#) or [Login](#)

 Search Tech.Pro

## • SHARE:

- 
- 
- 
- 
- 

- [Feed](#)
- [Tutorials](#)
- [Blogs](#)
- [Links](#)
- [Q&A](#)



The nerd table is now the cool table. [learn more](#)



**Eric Lippert**

posted 4 months ago *K*

[Benchmark C#](#)

# C# Performance Benchmark Mistakes, Part One



In this series of articles, I'm going to go through some of the mistakes I frequently see people making who are attempting to write *benchmarks* in C#. But before we get into the mistakes, I suppose I should introduce myself and define the term.

Hi, I'm Eric Lippert; I work at [Coverity](#) where I design and implement static analyzers to find bugs in C# programs. Before that I was at Microsoft for 16 years working on the C#, VBScript, JScript and Visual Basic compilers, amongst other things. I write a blog about language design issues at [EricLippert.com](#). Many thanks to the editors here at Tech.pro for inviting me to write this series.

OK, let's get into it. What exactly do I mean when I say *benchmark*?

The term comes originally from surveying; a surveyor would mark an object that was at an already-known position and then use that mark to determine the relative and absolute positions of other objects. In computing the term, like so much other jargon, has been liberally borrowed and now means pretty much any sort of performance comparison between two alternatives.

Benchmarking is often used to describe the performance of computer *hardware*: you write a program, compile and execute it on two different computers, and see which one performs better. That's not the kind of benchmarking I'm going to talk about in this series; rather, I want to talk about *performance benchmark tests for software*.

I want to clarify that further, starting with the meaning of "performance" itself. The first rule of software metrics is well known: *you get what you measure*.

If you reward people for making a measurable improvement in memory usage, don't be surprised if time performance gets worse, and vice versa. If you reward *improvement* rather than *achieving a goal* then you can expect that they'll keep trying to make improvements even after the goal has been achieved (or worse, even if it is never achieved!)

This brings us to our first benchmarking mistake:

## Mistake #1: Choosing a bad metric.

If you've chosen a bad metric then you're going to waste a lot of effort measuring and improving an aspect of the software that is not relevant to your users, so choose carefully.

For the rest of this series I'm going to assume that the relevant performance metric that your benchmark measures is *average execution time*, and not one of the hundreds of potential other metrics, like worst-case time, memory usage, disk usage, network usage, and so on. This is the most common metric for performance benchmarks and hence the one I see the most mistakes in.

I also want to clarify one other thing before we dive in. I'm assuming here that the purpose of the benchmark is to empirically determine the performance of a *small part* of a larger software project so that an *informed decision* can be made.

For example, you might have a program that, among its many other tasks, sometimes has to sort a large set of data. The benchmarks I'm talking about in this series are the narrowly targeted tests of, say, half a dozen different sort algorithms to determine which ones yield acceptable performance on typical data; I'm not talking about "end to end" performance testing of the entire application. Often in large software projects the individual parts have good performance in isolation, but bad performance in combination; you've got to test both.

That brings us to:

## Mistake #2: Over-focusing on subsystem performance at the expense of end-to-end performance.

This series of articles is going to be all about subsystem performance; don't forget to budget some time for end-to-end testing as well.

So far we've seen some very general mistakes; now let's start to dig into the actual mistakes people make in implementing and executing their subsystem performance benchmarks in C#. The number one most common mistake I see is, no kidding:

## Mistake #3: Running your benchmark in the debugger.

This is about the worst thing you can possibly do. The results will be totally unreliable. Think about all the things that are happening when you run a managed program in a debugger that are *not* happening when your customer runs the program: the CLR is sending information to the debugger about the state of the program, debug output is being displayed, heck, an entire other enormous process is running.

But it gets worse, far worse.

The jit compiler knows that a debugger is attached, and it deliberately de-optimizes the code it generates to make it easier to debug. The garbage collector knows that a debugger is attached; it works with the jit compiler to ensure that memory is cleaned up less aggressively, which can greatly affect performance in some scenarios.

But perhaps I am getting ahead of myself. What is this "jit compiler" thing? In order to make sense of the next episode in this series you'll need to have a pretty solid understanding of how compilation works in .NET. Here's the high level view.

Let's suppose you write some source code in C# using Visual Studio. When you build that project the IDE starts up the C# compiler. A compiler is by definition *a program which translates a program written in one language into "the same" program written in another language*. The C# compiler translates C# code into a different language, IL, the Intermediate Language. (Also sometimes notated CIL for Common IL or MSIL for Microsoft IL, but we'll just stick with "IL".)

IL is a very low-level language designed so that in its compressed binary form it is reasonably compact but also reasonably fast to analyze. A managed assembly (a .exe or .dll file) contains the IL for every method in the project as well as the "metadata" for the project: a compact description of all the classes, structs, enums, delegates, interfaces, fields, properties, methods, events,... and so on in your program.

When you run code in a managed assembly, the Common Language Runtime (CLR) reads the metadata out of the assembly to determine what the types and methods and so on are. But the real miracle of the CLR is the Just In Time compiler -- "the jitter" for short. The jitter is a compiler, so again, it translates from one language to another. The CLR runs the jitter on the IL associated with a method *immediately before that method is about to run for the first time* -- hence the name "Just In Time compiler". It translates the IL into the machine code that will actually execute on the processor.

So now perhaps it is more clear why understanding the jitter behaviour is so important when benchmarking code; the jitter is *dynamically generating the actual machine code on the fly*, and therefore determining how heavily optimized that machine code is. The jitter knows whether there is a debugger attached or not, and if there is then it figures it had better not be aggressive about optimizations because *you might be trying to inspect the code in the debugger*; heavily optimized code is harder to understand. But obviously the unoptimized code will be less performant, and therefore the benchmark is ruined.

Even if you don't run your benchmark program in the debugger it is still important to make sure that you are not telling the jitter to go easy on the optimizations:

## Mistake #4: Benchmarking the debug build instead of the release build.

If you compile a project in Visual Studio in "debug" mode then both the C# compiler and the jit compiler will again deliberately generate less-optimized code *even if you run the program outside of the debugger* on the assumption that clarity is better than speed when you are attempting to diagnose problems.

And of course the debug version of your program might contain special-purpose code of your own devising to make debugging easier. For example, expensive assertions might be checked which would be ignored in the release build.

Both mistakes #3 and #4 are actually specific versions of a more general mistake: testing the code in an environment radically different from the customer's environment. The customer is not going to be running the debug version of your product, so don't test that version. We'll come back to this point in a later episode.

Next time in this series I'll talk about mistakes made in specific measurement techniques; after that we'll take a look at some more subtle ways in which forgetting about the jitter can lead to bad benchmarks.

### Read related posts in this series:

- [C# Performance Benchmark Mistakes, Part Two](#)

Unlike  
this  
Like  
this  
Article




### Post a Comment

### Comments ()

-  pavan bangaram posted 4 months ago

Than x for providing this information..! This is so useful for me..!

[i Reply](#)

-  [Rohan Khan](#) posted 4 months ago  
A wonderful article and you have given a good description.You have described all points details.  
[i Reply](#)
-  [Joel Wilson](#) posted 4 months ago  
Good article; 2 thumbs up. Interestingly, we had a production problem just last week because of a missing method in a shared assembly. I did some testing to understand how/when the dynamic linkage kicks in and your mention of the JIT compiler in Mistake #3 confirm my empirical findings. Looking forward to the next article in the series.  
[i Reply](#)
-  [Ken Lee](#) posted 3 months ago  
I totally agree that using a debugger version of your IL is a mistake as a formal benchmark tool, but it can be useful in making comparisons between two different algorithms that do the same thing. If there is a substantial difference, that difference will probably be maintained in the release version.  
  
I recently compared three sorting algorithms, bubble sort, binary sort, and the built-in Array sort. As I expected the binary sort killed bubble and was beat by the built-in sort. I was pleasantly surprised that the rate of slow-down with size increase matched between my and built-in sorts and remained consistent in comparison in both debug and release.  
  
Memory stress to failure is easier to reach in debug mode as well. This may give some insight on how well your release version will perform in a lower memory device.  
  
[i Reply](#)

Likes

8

About the Author

 [Eric Lippert](#)  
[Follow](#)

Views  
17k

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

Want to write with us?

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

Write & Win

Tech.Pro Daily Email

Get the latest content from Tech.Pro delivered to your Inbox each day. Simply create an account to subscribe.

Subscribe

Copyright © 2013  
Tech.Pro

- [Feed](#)
- [Tutorials](#)
- [Blogs](#)
- [Links](#)
- [Q&A](#)

- [People](#)
- [Profiles](#)
- [Authors](#)
- [Projects](#)
- [My Network](#)

- [More](#)
- [How TechPro Works](#)
- [About Us](#)
- [Contact Us](#)
- [Site Map](#)

- [Blog](#)
- [Twitter](#)
- [Facebook](#)