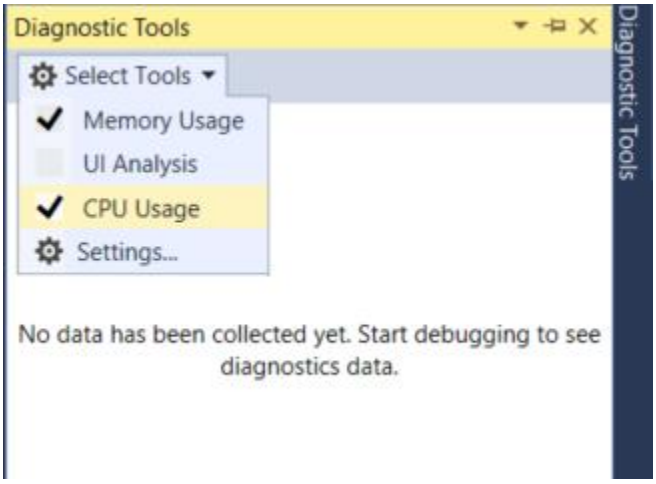
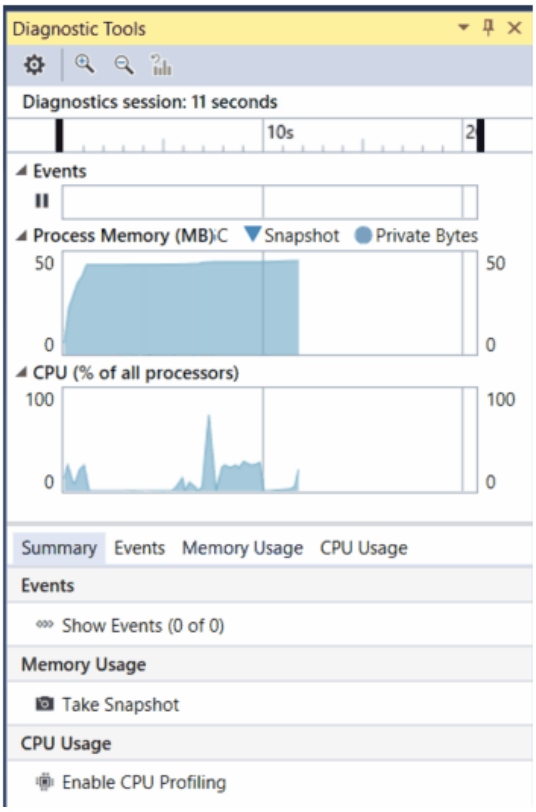
Visual Studio Profiling Tools

If your app runs too slowly or uses too much memory, you may need to test your app with the profiling tools early on. Visual Studio provides a variety of profiling tools to help you diagnose different kinds of performance issues depending on your app type.

Measure performance while debugging

The profiling tools that you can access during a debugging session are available in the Diagnostic Tools window. The Diagnostic Tools window appears automatically unless you have turned it off. To bring up the window, click **Debug / Windows / Show Diagnostic Tools**. With the window open, you can select tools for which you want to collect data.





While you are debugging, you can use the **Diagnostic Tools** window to analyze CPU and memory usage, and you can view events that show performance-related information.

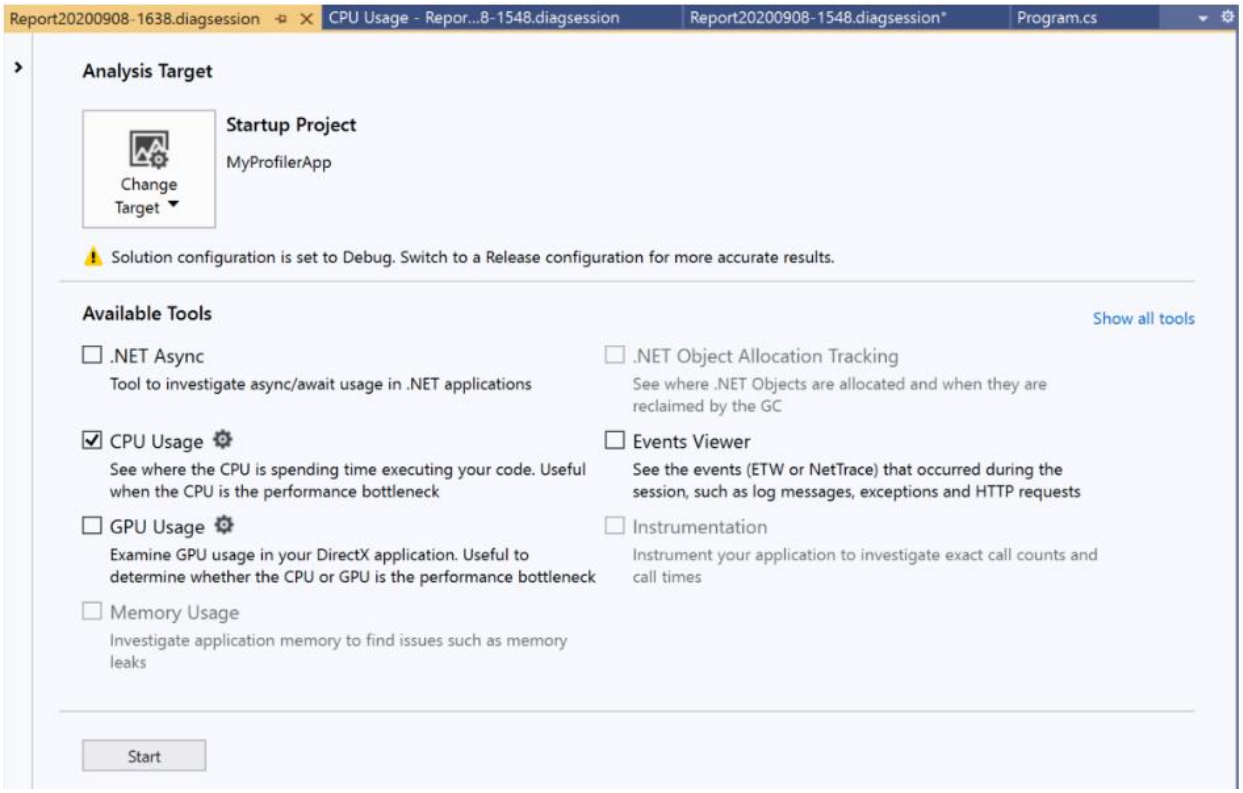
The Diagnostic Tools window is a common way to profile apps, but for Release builds you can also do a post-mortem analysis of your app instead. Tools available in the Diagnostic Tools window or during a debugging session include:

- CPU usage
- Memory usage
- PerfTips

Measure performance in release builds

Tools in the Performance Profiler are intended to provide analysis for **Release** builds. In the Performance Profiler, you can collect diagnostic info while the app is running, and then examine the collected information after the app is stopped (a post-mortem analysis).

Open the Performance Profiler by choosing **Debug** > **Performance Profiler** (or **Alt + F2**).

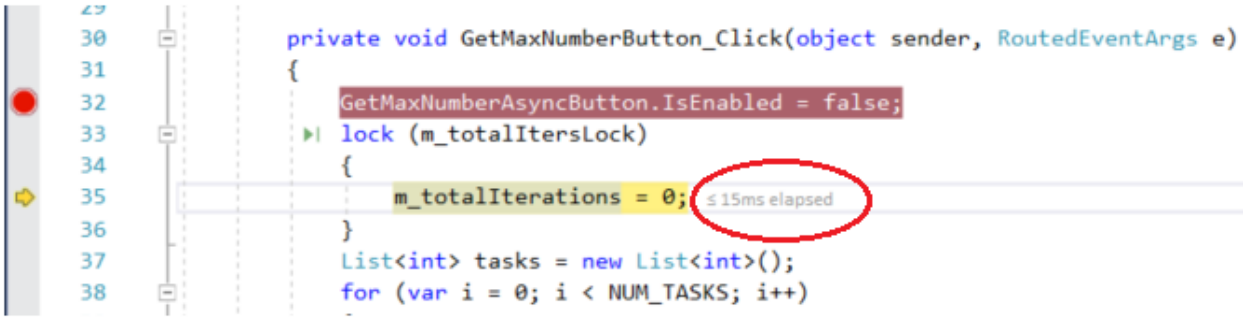


Tools available in the Performance Profiler include:

- CPU usage
- .NET object allocation
- Memory usage
- .NET async tool
- Database tool
- GPU usage

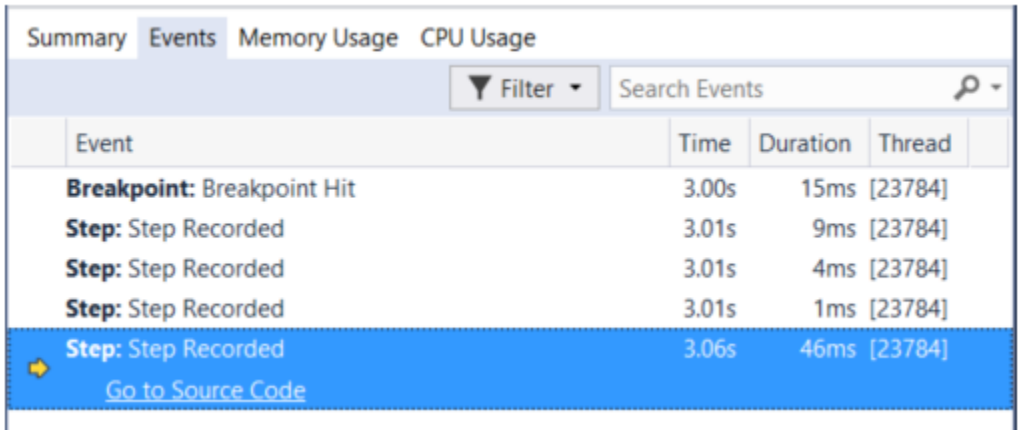
Examine performance using PerfTips

Often, the easiest way to view performance information is to use PerfTips. Using PerfTips, you can view performance information while interacting with your code. You can check information such as the duration of the event (measured from when the debugger was last paused, or when the app started). For example, if you step through code (F10, F11), PerfTips show you the app runtime duration from the previous step operation to the current step.



You can use PerfTips to examine how long it takes for a code block to execute, or how long it takes for a single function to complete.

PerfTips show the same events that also show up in the **Events** view of the Diagnostic Tools. In the **Events** view, you can view different events that occur while you are debugging, such as the setting of a breakpoint or a code stepping operation.



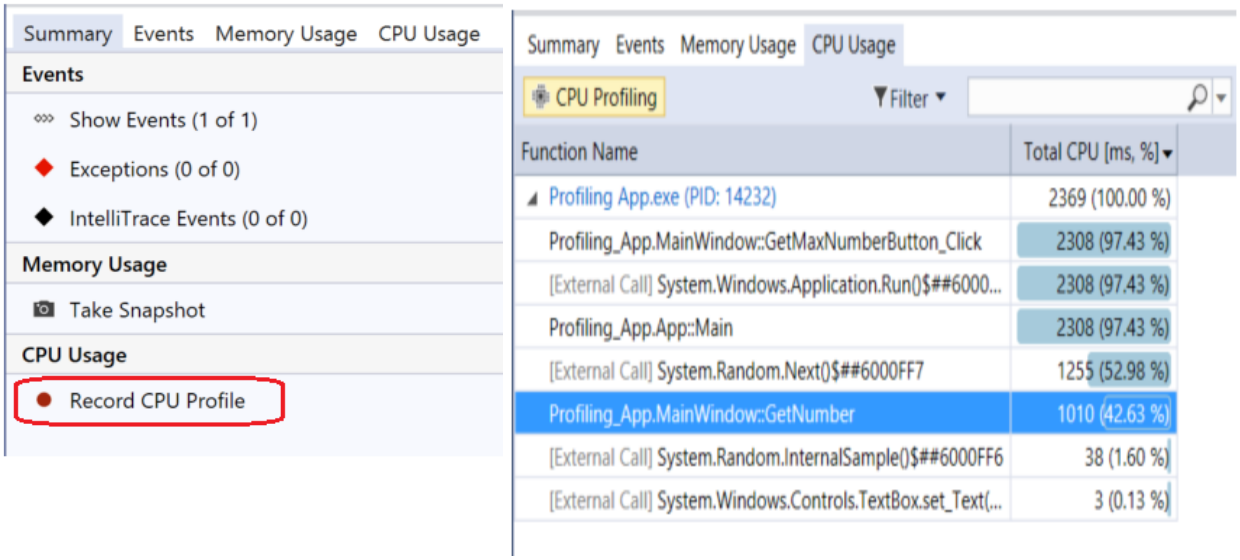
Analyze CPU usage

The CPU Usage tool is a good place to start analyzing your app's performance. It will tell you more about CPU resources that your app is consuming. You can use the debugger-integrated CPU Usage tool or the post-mortem CPU Usage tool.

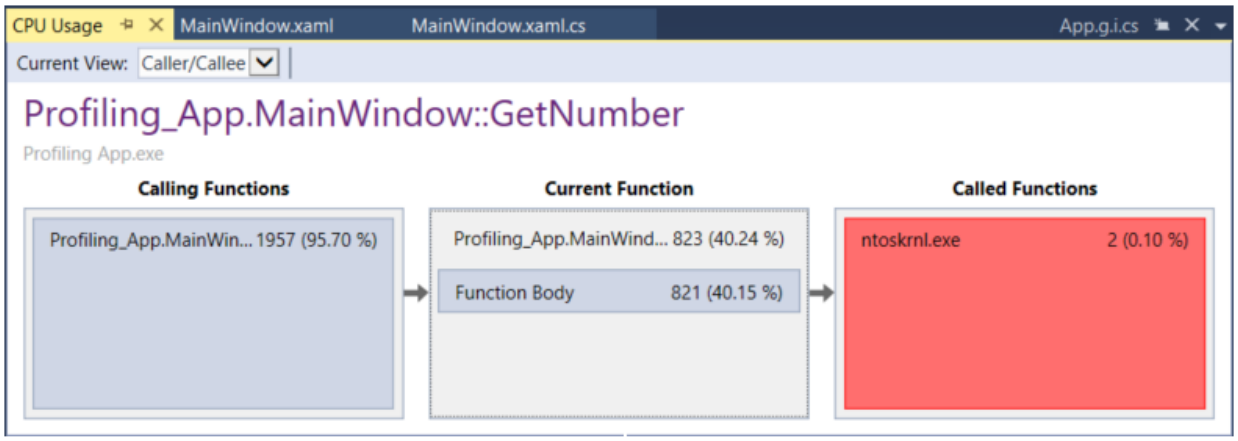
When using the debugger-integrated CPU Usage tool, open the **Diagnostics Tool** window (if it's closed, choose **Debug / Windows / Show Diagnostic Tools**). While debugging, open the **Summary view**, and select **Record CPU Profile**.

One way to use the tool is to set two breakpoints in your code, one at the beginning and one at the end of the function or the region of code you want to analyze. Examine the profiling data when you are paused at the second breakpoint.

The **CPU Usage** view shows you a list of functions ordered by longest running, with the longest running function at the top. This can help guide you to functions where performance bottlenecks are happening.



Double-click on a function that you are interested in, and you will see a more detailed three-pane "butterfly" view, with the selected function in the middle of the window, the calling function on the left, and called functions on the right. The **Function Body** section shows the total amount of time (and the percentage of time) spent in the function body excluding time spent in calling and called functions. This data can help you evaluate whether the function itself is a performance bottleneck.



Analyze memory usage

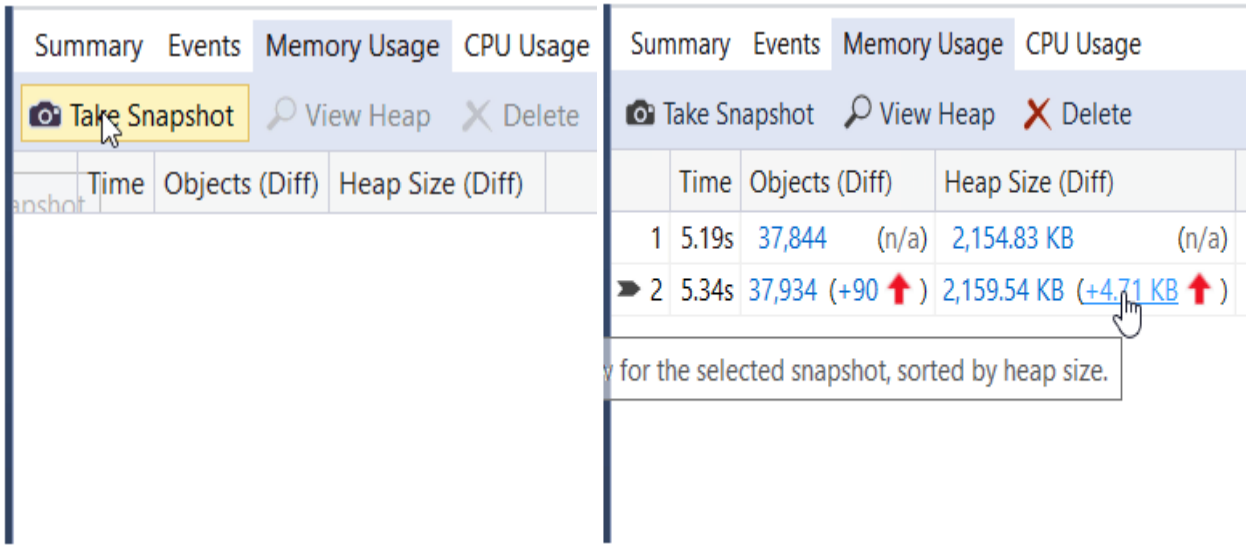
The **Diagnostic Tools** window also allows you to evaluate memory usage in your app using the **Memory Usage** tool. For example, you can look at the number and size of objects on the heap. You can use the debugger-integrated Memory Usage tool or the post-mortem Memory Usage tool in the Performance Profiler.

.NET developers may choose between either the .NET Object Allocation tool or the Memory usage tool.

- The .NET Object Allocation tool helps you identify allocation patterns and anomalies in your .NET code, and helps identify common issues with garbage collection. This tool runs only as a post-mortem tool. You can run this tool on local or remote machines.

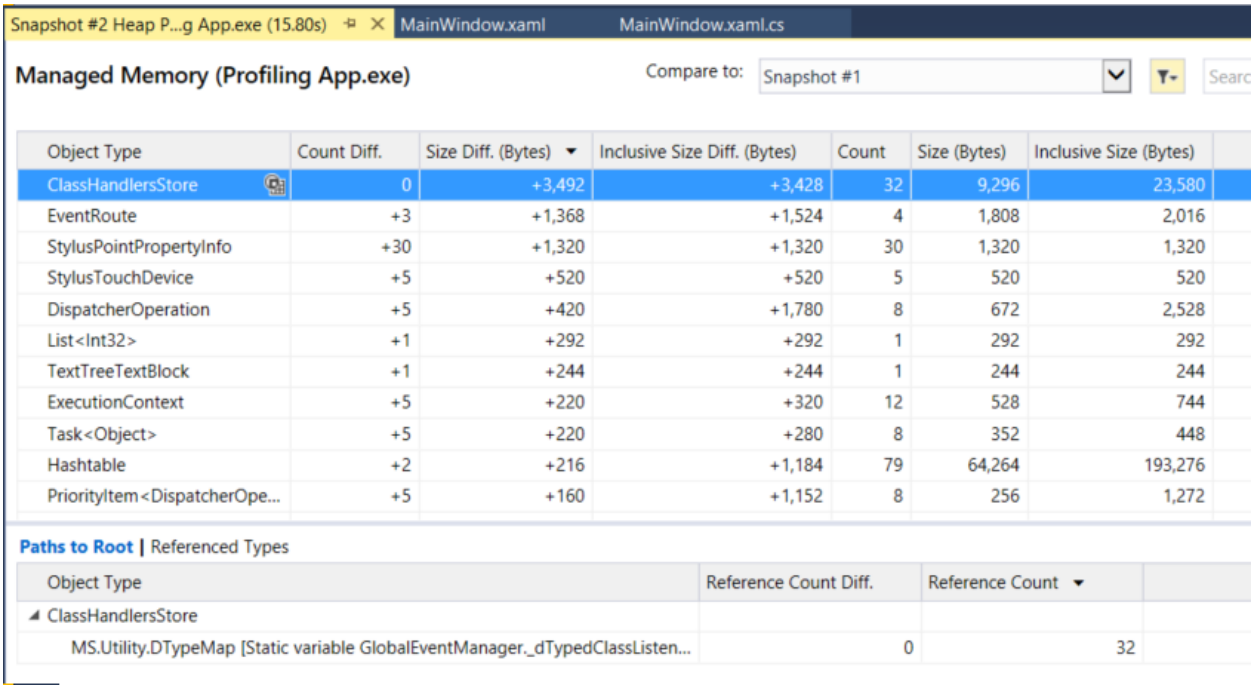
- The Memory usage tool is helpful in identifying memory leaks, which are not typically common in .NET apps. If you need to use debugger features while checking memory, such as stepping through code, the debugger-integrated Memory usage tool is recommended.

To analyze memory usage with the Memory Usage tool, you need to take at least one memory snapshot. Often, the best way to analyze memory is to take two snapshots; the first right before a suspected memory issue, and the second snapshot right after a suspected memory issue occurs. Then you can view a diff of the two snapshots and see exactly what changed. The following illustration shows taking a snapshot with the debugger-integrated tool.



When you select one of the arrow links, you are given a differential view of the heap (a red up arrow ⬆ shows an increasing object count (left) or an increasing heap size (right)). If you click the right link, you get a differential heap view ordered by objects that increased the most in heap size. This can help you pinpoint memory problems. For example, in the illustration below, the bytes used by ClassHandlersStore objects increased by 3,492 bytes in the second snapshot.

If you click the link on the left instead in the **Memory Usage** view, the heap view is organized by object count; the objects of a particular type that increased the most in number are shown at the top (sorted by **Count Diff** column).


Snapshot #2 Heap P...g App.exe (15.80s)
MainWindow.xaml
MainWindow.xaml.cs
Managed Memory (Profiling App.exe)
Compare to:
Snapshot #1
Object Type
Count Diff.
Size Diff. (Bytes)
Inclusive Size Diff. (Bytes)
Count
Size (Bytes)
Inclusive Size (Bytes)
ClassHandlersStore 0 +3,492 +3,428 32 9,296 23,580
EventRoute +3 +1,368 +1,524 4 1,808 2,016
StylusPointPropertyInfo +30 +1,320 +1,320 30 1,320 1,320
StylusTouchDevice +5 +520 +520 5 520 520
DispatcherOperation +5 +420 +1,780 8 672 2,528
List<Int32> +1 +292 +292 1 292 292
TextTreeTextBlock +1 +244 +244 1 244 244
ExecutionContext +5 +220 +320 12 528 744
Task<Object> +5 +220 +280 8 352 448
Hashtable +2 +216 +1,184 79 64,264 193,276
PriorityItem<DispatcherOpe... +5 +160 +1,152 8 256 1,272
Paths to Root | Referenced Types
Object Type
Reference Count Diff.
Reference Count
ClassHandlersStore
MS.Utility.DTypeMap [Static variable GlobalEventManager._dTypedClassListen... 0 32