

Getting Started with dotTrace Performance

© 2010, JetBrains s.r.o



Getting Started with dotTrace Performance

Getting Started with dotTrace Performance

© 2010, JetBrains s.r.o

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

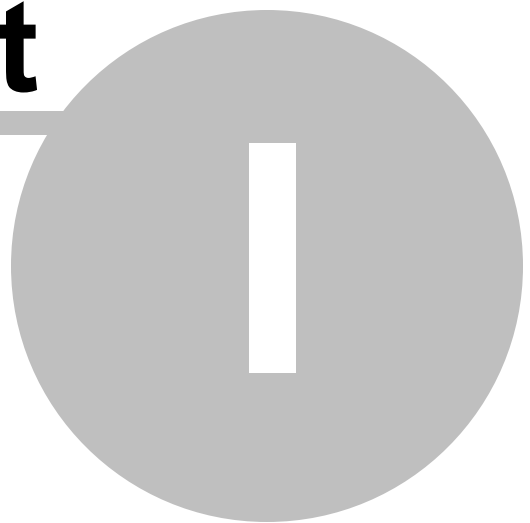
Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Table of Contents

| | |
|--|-----------|
| Foreword | 5 |
| Part I Basics | 7 |
| 1 Introduction..... | 7 |
| 2 Launching dotTrace Performance..... | 7 |
| 3 Grabbing Snapshots with Controller Panel..... | 8 |
| 4 Views..... | 10 |
| 5 Profiling Options..... | 15 |
| 6 Options..... | 17 |
| Part II Profiling Applications | 25 |
| 1 Types of Applications..... | 26 |
| Standalone Applications | 26 |
| Silverlight Applications | 27 |
| Web Application using WebDev Server | 28 |
| Web Application using Internet Information Service | 29 |
| Windows Services | 30 |
| .NET Process | 31 |
| Smart Devices (Standalone Application) | 32 |
| 2 Remote Profiling..... | 34 |
| 3 Working with Source Code..... | 35 |
| 4 Locating Functions..... | 37 |
| 5 Annotating Functions..... | 38 |
| 6 Adjusting Call Times..... | 40 |
| 7 Working with Tabs..... | 42 |
| 8 Flattening Recursive Calls..... | 43 |
| 9 Interpreting Call Information..... | 44 |
| 10 Comparing Snapshots..... | 45 |

Part



1 Basics

In this section you can learn the basics of dotTrace Performance and how to interpret the information.

1.1 Introduction

Welcome to the Step by Step guides for dotTrace Performance 4.0.



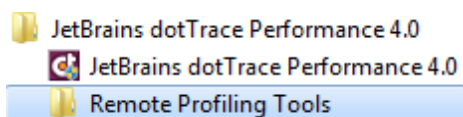
In this guide you can step through the different operations you can perform with dotTrace Performance 4.0 and with clear concise instructions you can get up to speed in no time.

If you have not yet downloaded dotTrace Performance, go to <http://www.jetbrains.com/profiler> to get the latest version

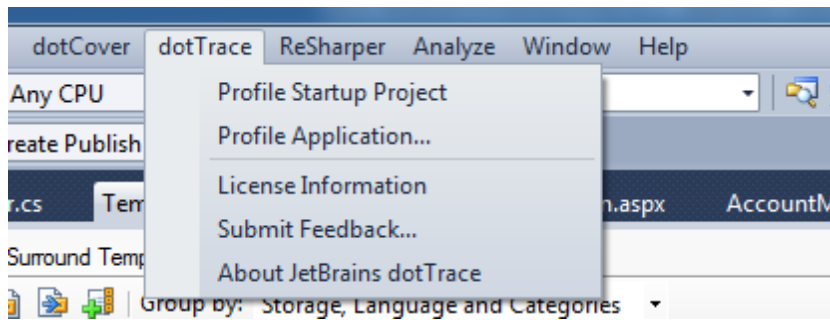
1.2 Launching dotTrace Performance

dotTrace Performance can be launched in two ways: Standalone or via Visual Studio.

- **Standalone.** This mode is ideal for application where the original project or source code is not always available or when we want to run dotTrace without having Visual Studio installed or launched. dotTrace installs in the Start Menu for easy access. You can also launch it directly from the installation menu.



- **Visual Studio.** This mode is ideal for when we want to profile an existing project running under Visual Studio. In order to be able to run dotTrace, open a project and select Profile from the dotTrace menu installed inside Visual Studio

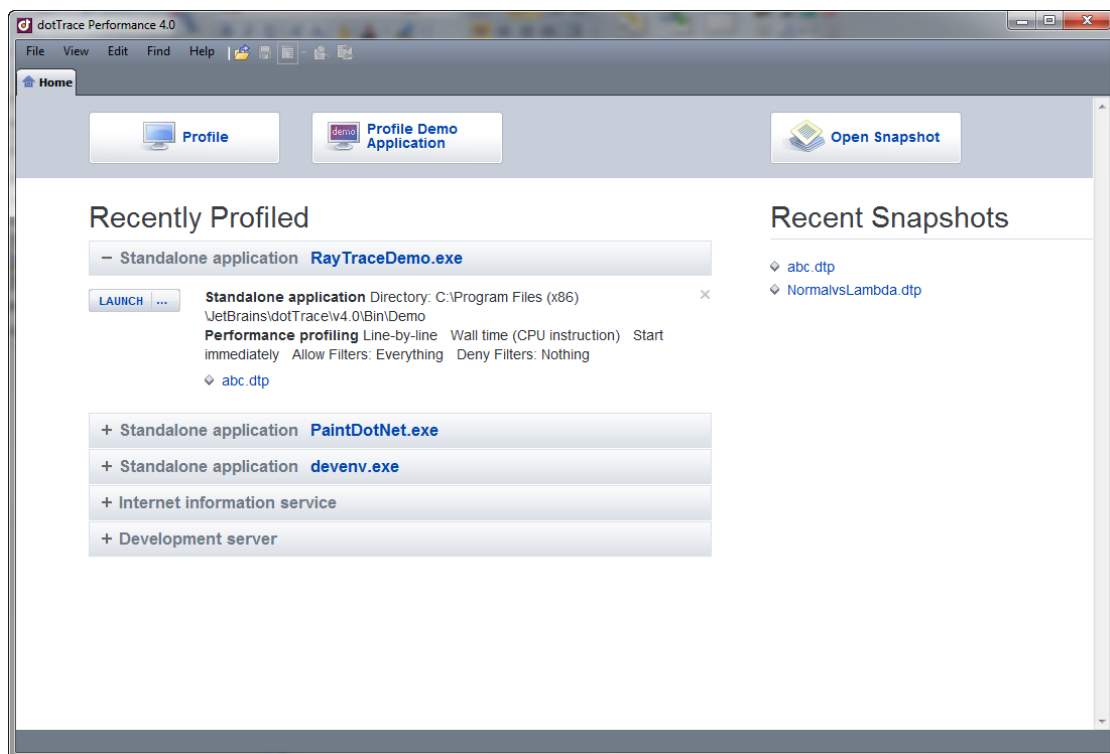


1.3 Grabbing Snapshots with Controller Panel

In this section we will see how to grab snapshots for profiling analysis. Snapshots are captures of key performance counters of applications, which can include information like number of calls made by functions, duration of these calls, percentage taken up in respect to the overall application, etc.

In order to profile an application, we need either the original Visual Studio project or the executable (see [Launching dotTrace Performance](#) for more information). In this case we are going to launch the demo Profiling application that comes with dotTrace Performance.

1. Launch dotTrace Performance Standalone from the Start Menu.

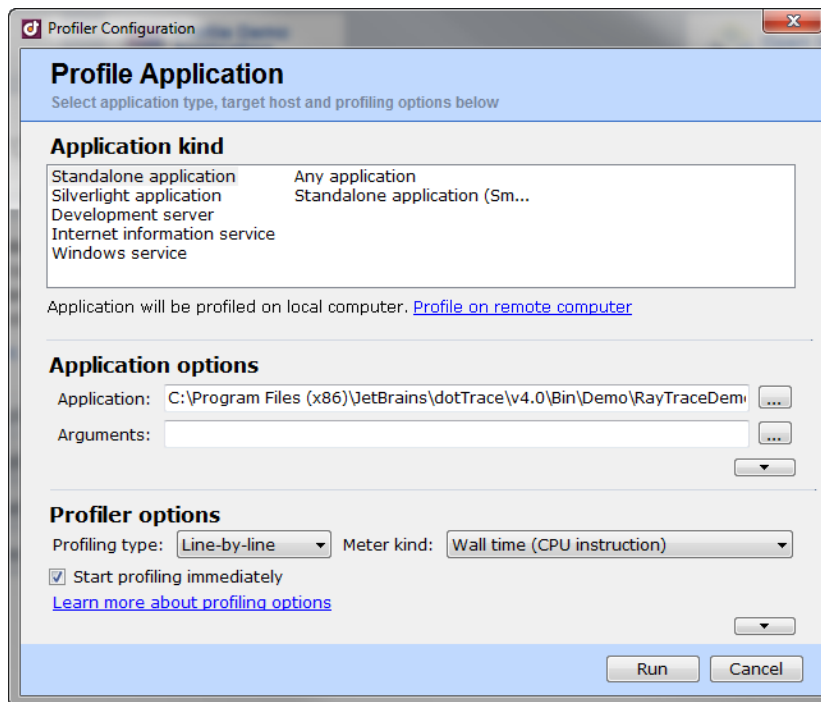


2. There are two ways we can profile the demo application:

- a) Setting the properties manually by selecting the **Profile** button.
- b) Using the predefined properties by selecting the **Profile Demo Application** button.

We are going to choose option b).

- Click on **Profile Demo Application** and we will be presented with a Profile Application dialog box. In this dialog box, we can define the type of application we want to profile (Standalone, Web Application, Windows Service, Silverlight, etc.), the type of profiling we want to do, as well as other options which will be discussed later on. For now let's leave default options and click **Run**.



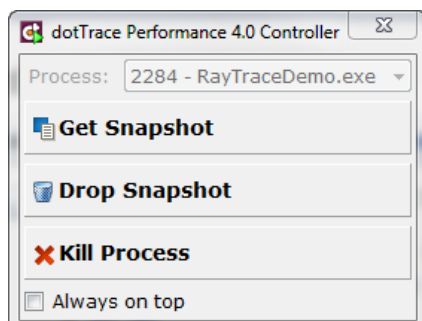
- dotTrace Performance will now launch the dotTrace Controller Panel as well as the application being profiled. The Controller has three functions:

Get Snapshot: Gets a snapshot of the current profiling data.

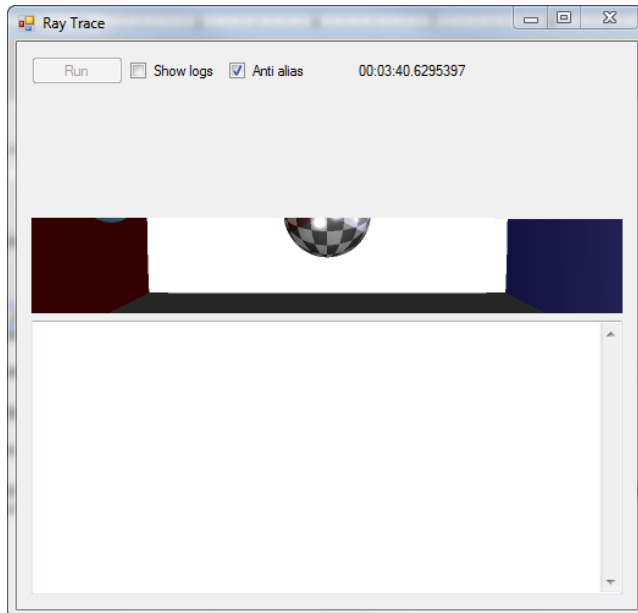
Drop Snapshot: Discards current profiling data

Kill Process: Kills the application being profiled (same as closing the application)

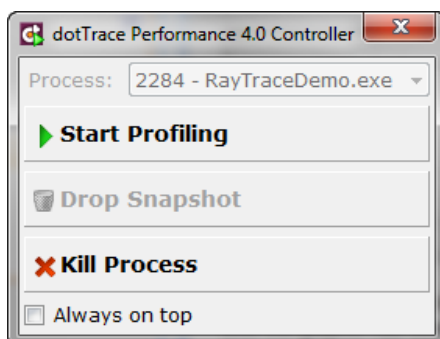
The profiler is now running in the background gathering information. At any point, we can grab a snapshot of the activity by clicking on **Get Snapshot**. This will capture the information and launch dotTrace Performance to display it. The Controller will remain active, allowing us to start profiling again (the **Get Snapshot** button now changes to **Start Profiling**).



5. The sample application draws vector graphics on the screen. In order to find potential bottlenecks and get some valid profiling data, check the **Anti Alias** checkbox and then click on the **Run** button of the application and wait until the graphic has been painted (The smaller we make the application window before hitting Run, the less time it will take to complete).



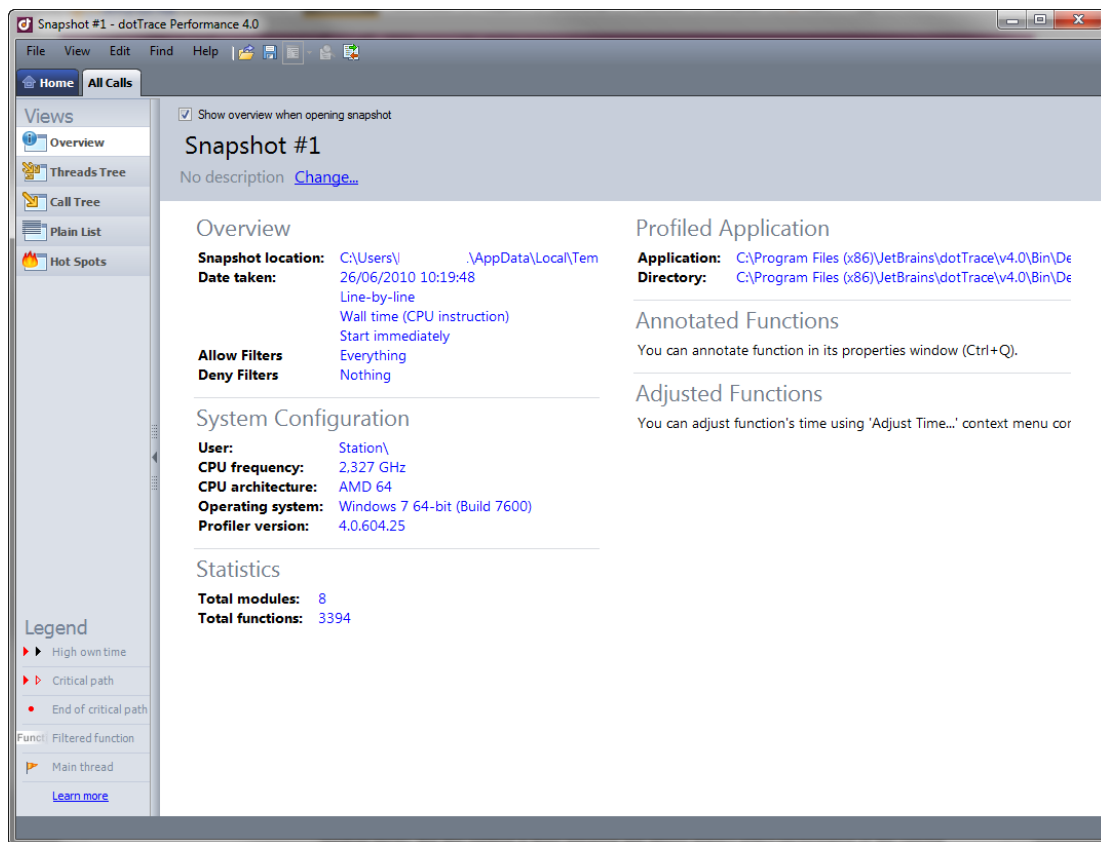
6. Once the process has completed, we can click on the **Get Snapshot** button. The Controller window should now change to display the **Start Profiling** button.



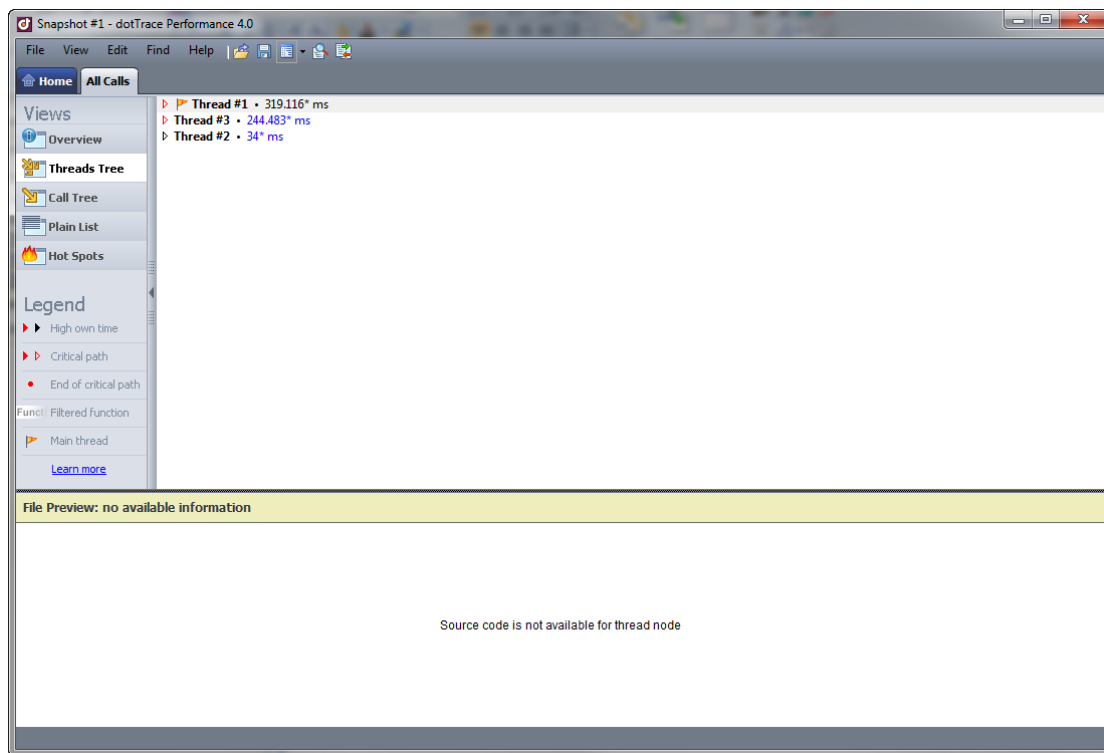
1.4 Views

dotTrace Performance can display different representations of the same profiling data. In this section we will see what each of these are.

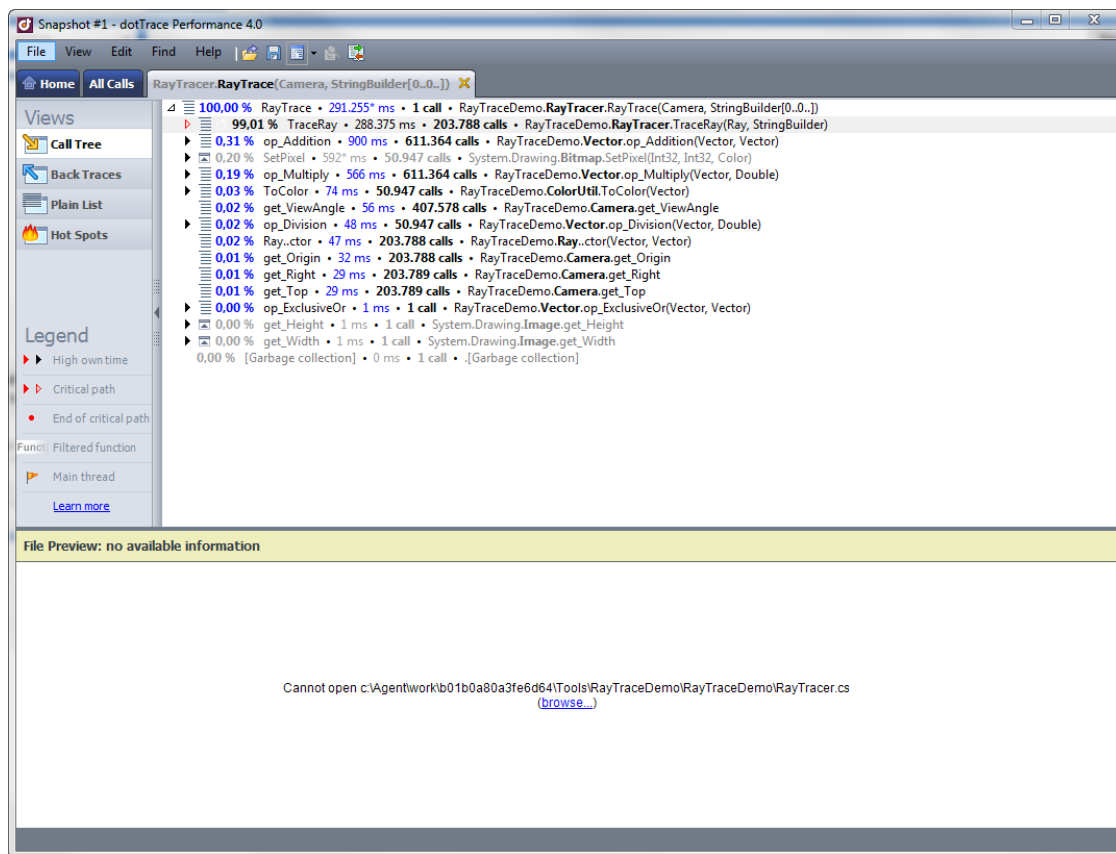
- **Overview:** Represents information about the System Configuration, as well as the specific profiling settings such as type of profiling, type of application, etc.



- **Threads Tree:** By default it displays the Threads Tree. This view represents all the different threads that are running in the application, along with the time each one has run. The main application thread is denoted with a flag. The lower part of the screen is reserved for showing the source code corresponding to the application.



- **Call Tree:** Displays how calls are made from one method to another. The calls are grouped by the percentage of the overall time they represent. In this view as well as other views, some calls are greyed out, as these correspond to the filtered function. While profiling any application, there are calls to one's own code as well as calls to the .NET framework BCL and third-party libraries. dotTrace allows you to filter out some of these calls and concentrate only on the application code itself. This is what filtering allows you to do.



- **Plain List:** Displays a list of function calls. These can optionally be grouped by Class or Namespace. The list displays the Time the function has taken, how much of that time is spent in the function itself, as opposed to other functions it calls, as well as the number of calls made to the function. There is also a column representing the number of recursive calls made, where applicable.

The screenshot shows the dotTrace Performance 4.0 interface. The top menu bar includes File, View, Edit, Find, and Help. Below the menu is a toolbar with icons for various actions. The left sidebar contains a 'Views' section with options: Overview, Threads Tree, Call Tree, Plain List, and Hot Spots. The 'Hot Spots' view is currently selected. Below the sidebar is a 'Legend' section with icons for 'High own time', 'Critical path', and 'End of critical path'. The main area displays a table of functions called by RayTraceDemo.Program.Main. The table has columns for Function Name, Time, ms, Own Time, ms, Calls, and Rec. Calls. The functions listed include RayTraceDemo.RayTracer.RayTrace, RayTraceDemo.RayTracer.TraceRay, RayTraceDemo.RayTracer.IntersectRay, RayTraceDemo.Sphere.Intersect, RayTraceDemo.Vector.op_Multiply, RayTraceDemo.Plane.Intersect, RayTraceDemo.Vector.get_Z, RayTraceDemo.Vector.get_X, RayTraceDemo.Vector.get_Y, RayTraceDemo.Vector.op_Subtraction, and RayTraceDemo.Vector.op_MultiplyDouble. Below the table, there is a section titled 'Functions called by RayTraceDemo.Program.Main' which lists the same functions with their respective times and call counts. At the bottom, there is a 'File Preview' section with the message 'no available information' and a link to 'Cannot open c:\Agent\work\lb01b0a80a3fe6d64\Tools\RayTraceDemo\RayTraceDemo\Program.cs (browse...)'. The status bar at the bottom indicates '© 2010, JetBrains s.r.o'.

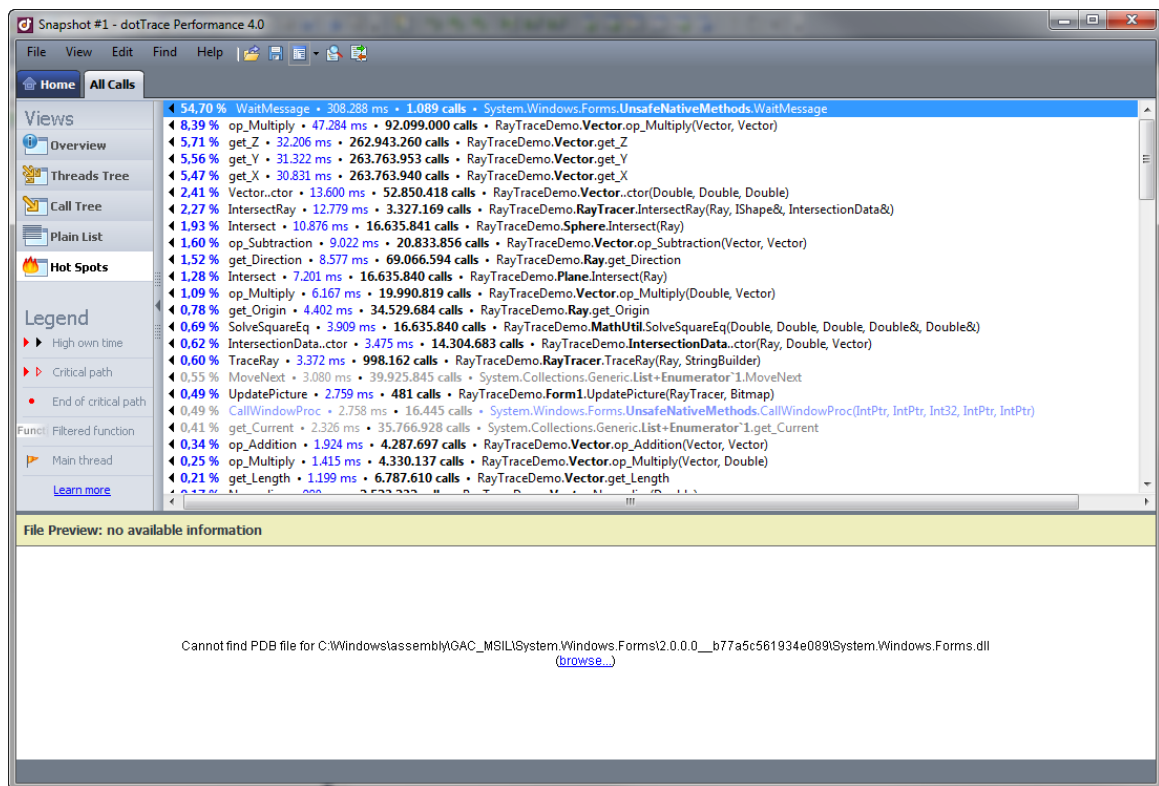
| Function Name | Time, ms | Own Time, ms | Calls | Rec. Calls |
|--|----------|-------------------|-------------|------------|
| RayTraceDemo.RayTracer.RayTrace(Camera, StringBuilder[0..0..]) | 291.255 | 504 (0,17 %) | 1 | 0 |
| RayTraceDemo.RayTracer.TraceRay(Ray, StringBuilder) | 288.375 | 4.160 (1,44 %) | 1.221.338 | 0 |
| RayTraceDemo.RayTracer.IntersectRay(Ray, IShape&, IntersectionData&) | 239.436 | 15.521 (6,48 %) | 4.071.101 | 0 |
| RayTraceDemo.Sphere.Intersect(Ray) | 156.422 | 12.907 (8,25 %) | 20.355.505 | 0 |
| RayTraceDemo.Vector.op_Multiply(Vector, Vector) | 133.345 | 56.854 (42,64 %) | 113.724.034 | 0 |
| RayTraceDemo.Plane.Intersect(Ray) | 59.048 | 8.495 (14,39 %) | 20.355.505 | 0 |
| RayTraceDemo.Vector.get_Z | 38.679 | 38.679 (100,00 %) | 325.096.941 | 0 |
| RayTraceDemo.Vector.get_X | 37.195 | 37.195 (100,00 %) | 326.091.264 | 0 |
| RayTraceDemo.Vector.get_Y | 36.664 | 36.664 (100,00 %) | 326.091.254 | 0 |
| RayTraceDemo.Vector.op_Subtraction(Vector, Vector) | 35.984 | 11.142 (30,96 %) | 25.593.417 | 0 |
| RayTraceDemo.Vector.op_Multiply(Double, Vector) | 22.478 | 7.401 (32,93 %) | 24.552.175 | 0 |

| Function Name | Time, ms | Own Time, ms | Calls | Rec. Calls |
|--|----------|--------------|-------|------------|
| System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32, ApplicationContext) | 335.352 | 5 (0,00 %) | 1 | 0 |
| RayTraceDemo.Form1..ctor | 1.534 | 20 (1,27 %) | 1 | 0 |
| System.BadImageFormatException..ctor | 349 | 2 (0,43 %) | 2 | 0 |
| System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(Boolean) | 26 | 9 (35,42 %) | 1 | 0 |
| System.Windows.Forms.Application.EnableVisualStyles | 19 | 4 (21,87 %) | 1 | 0 |
| System.Windows.Forms.Application.Run(Form) | 1 | 1 (99,71 %) | 1 | 0 |
| System.Runtime.Versioning.VersioningHelper..ctor | 0 | 0 (100,00 %) | 1 | 0 |
| System.Windows.Forms.Application..ctor | 0 | 0 (83,01 %) | 1 | 0 |

File Preview: no available information

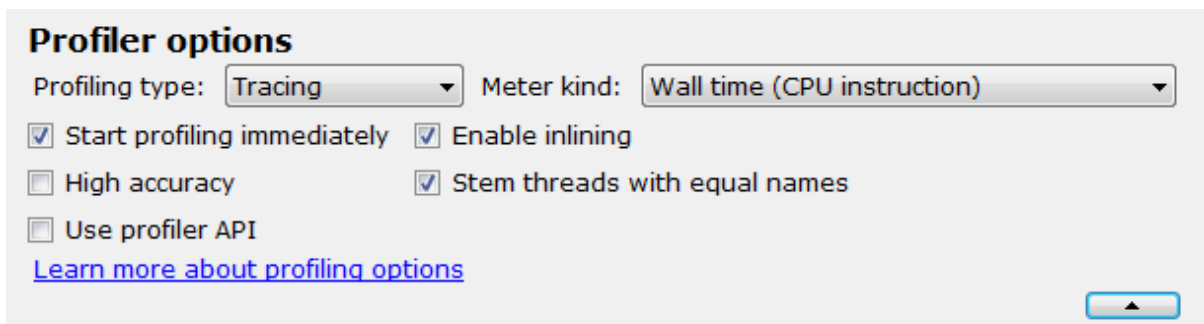
Cannot open c:\Agent\work\lb01b0a80a3fe6d64\Tools\RayTraceDemo\RayTraceDemo\Program.cs (browse...)

- **Hot Spots:** Displays potential performance issues. The view is representing the call stack from in to out, that is, the root is the last call made. By expanding each node, it displays a wall up the stack.



1.5 Profiling Options

Depending on the type of application we want to profile and the results we need to obtain from it, the profiling options we choose may differ. dotTrace Performance provides us with multiple choices when it comes to profiling options. In this section we will examine all the different possibilities.

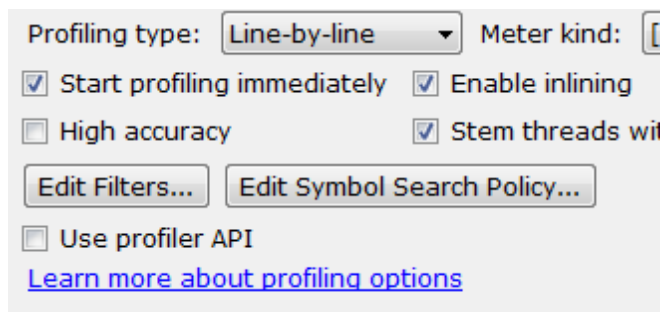


Profiling Type

Profiling Type indicates the way we want our application to be profiled in regard to the analysis information captured. The resulting snapshot may differ significantly based on the type of profiling we choose. It is important to make sure we select the type that is most appropriate for our needs.

- **Tracing:** In Tracing mode dotTrace Performance receives notifications from the CLR on every entry to and exit from a method. At each event, the current time is measured and the results are summarized in the snapshot. In this mode we obtain precise timing information and numbers of calls. dotTrace Performance architecture makes this method a reasonable compromise between amount of information obtained and the performance impact due to profiling.
- **Sampling:** In Sampling mode dotTrace Performance stops all threads every so often and grabs current call stack information on them. The output snapshot reflects statistics of the observed call stacks. In this mode we do not obtain numbers of calls and the times are approximate, since they cannot have accuracy more than the sampling frequency which is several hundred hertz. This type is least intrusive and almost has no affect on the performance of the application.
- **Line-by-Line:** In Sample mode dotTrace Performance collects timing information for every statement in methods where source code is available. This allows us to collect more detailed information for methods which perform significant algorithmic work. We can specify exactly which methods we want to be profiled or profile all methods for which dotTrace Performance can locate symbol information.

When selecting Line-By-Line Profiling, there are two more options we can specify:



Edit Filters allow us to specify exactly which functions you want to be profiled line-by-line.

Edit Symbol search Policy lets us configure places where profiler will look for symbol files for modules of profiled application.

Meter Kind

Meter Kind indicates the technique used by dotTrace Performance for obtaining the current time. Similar to Profiling Type, there are various options we can choose from depending on our needs.

- **Wall time (CPU Instruction):** This is the simplest and fastest way to measure wall time (that is, the time we observe on a wall clock). However, on some older multi-core processors this may produce incorrect results due to the cores timers being desynchronized. If this is the case, it is recommended to use Performance Counter.
- **Wall time (Performance Counter):** Performance counters is part of the Windows API and it allows taking time samples in a hardware-independent way. However, being an API call, every measure takes substantial time and therefore has an impact on the profiled application.
- **Thread time:** In a multi-threaded application concurrent threads contribute to each other's wall time. To avoid such interference we can use thread time meter which ,makes system API calls to get the amount of time given by the OS scheduler to the thread. The downsides are that taking thread time samples is much slower than using CPU counter and the precision is also limited by the size of quantum used by thread scheduler (normally 10ms). This mode is only supported when the **Profiling Type** is set to **Sampling**

Other Options

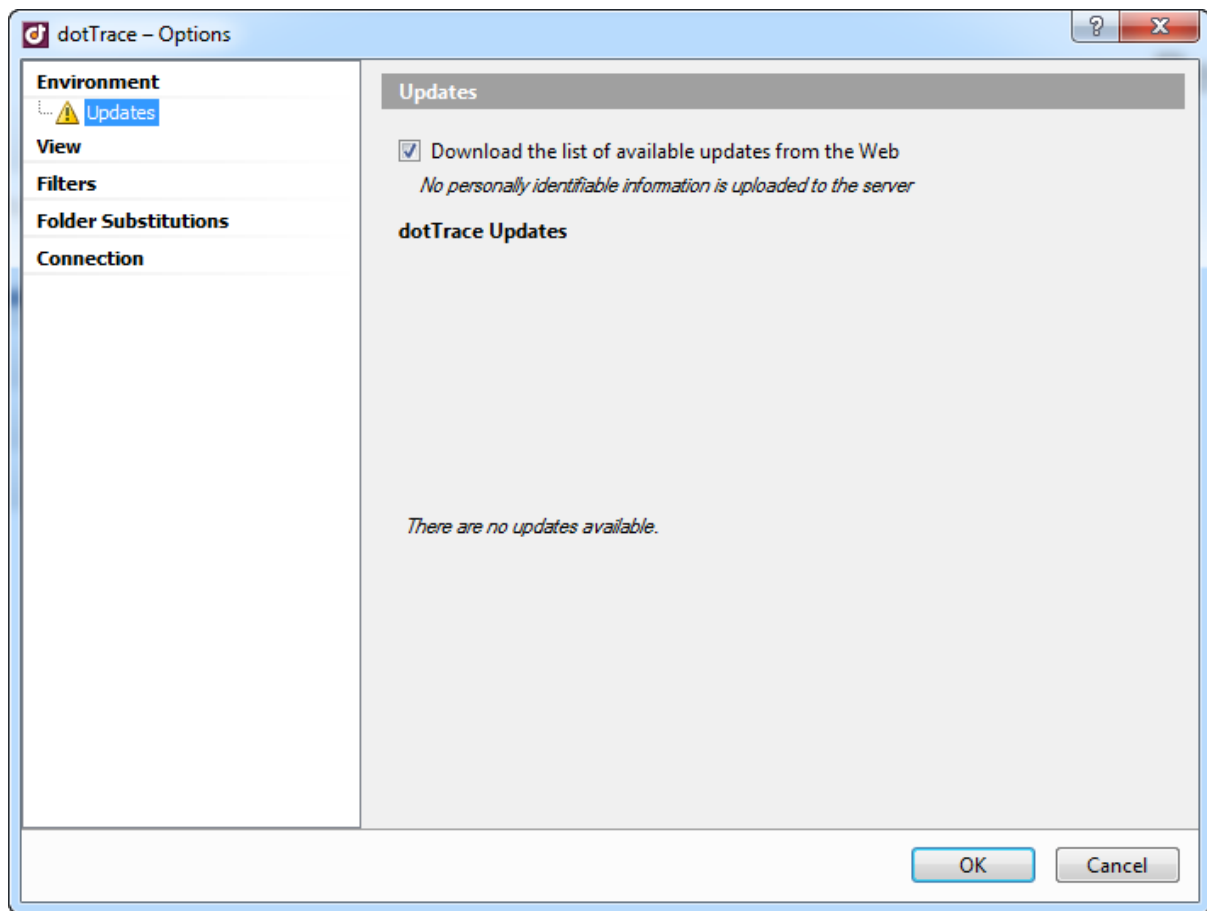
- **Start Profiling Immediately:** When running dotTrace Performance in Standalone, we can check this option to start profiling as soon as we hit **Run**.
- **High Accuracy:** This option improves accuracy by taking more time samples to take into account time spent in profiler itself. The price is that enabling this option increases application slow-down. Normally enabling this option makes sense if you have a lot of short-running functions.
- **Enable Inlining:** This option can be used to disable method inlining done by JIT. This way you get callstacks which closely resemble your program structure.
- **Stem threads with equal names:** When this option is on, profiler merges threads into single node if their names are equal.

1.6 Options

dotTrace Performance offers a series of Application Options to work with which allow us to customize certain behaviors and appearances. Under the **View | Options** menu we can access the following global settings

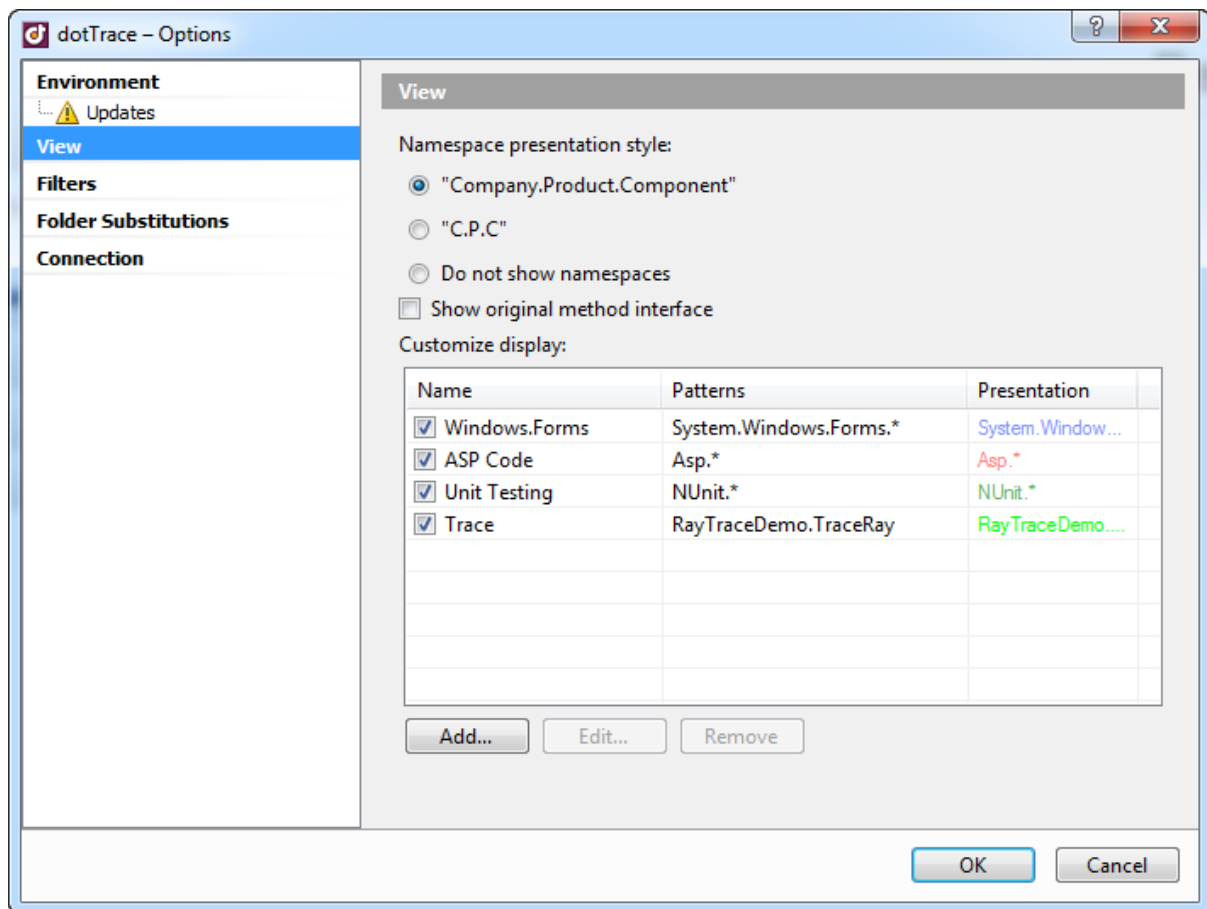
Auto Update

We can indicate to dotTrace Performance whether we want it to perform automatic checks for newer versions



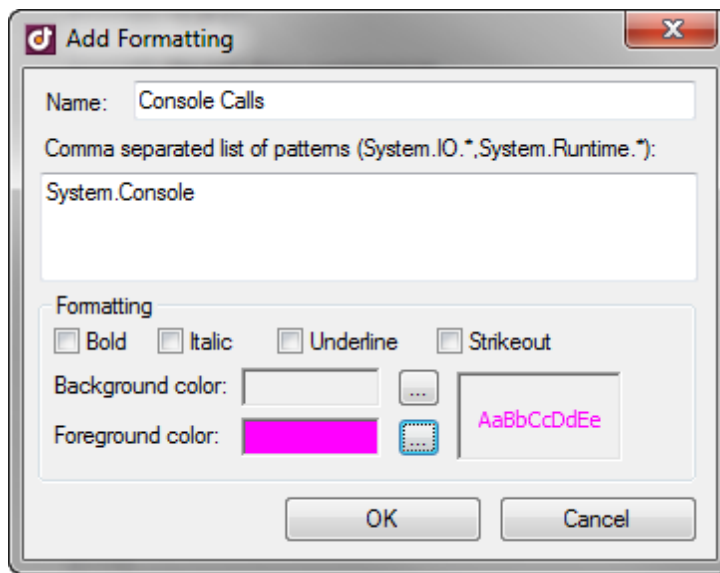
Views

dotTrace Performance provides us with the ability to fine-tune how we want certain namespaces to appear in the different Views.



Views by default use two main colors to depict function calls: Black for non-filtered functions and Grey for filtered ones. In addition, functions are by default represented using the Company.Product.Component nomenclature. Often times we might want to provide additional coloring to make it easier for specific namespaces to be highlighted or change the naming style. We can do both of these via the **View** settings in the **Options** dialog.

1. The **View** section allows us to firstly change the Namespace presentation style. We can choose one of three:
 - a) Company.Product.Component
 - b) C.P.C
 - c) Do not show
2. In order to customize the colors, we need to add a new formatting. To do so, we need to click on the Add button.
3. In the dialog box that displays, we need to provide a name for the formatting, in this case lets enter Console Calls and type System.Console in the pattern box.



We can select any foreground or background color, as well as indicate whether we want the calls to be Bold, Italic, etc.

4. We click the **OK** button to save the new formatting options.

Once the formatting has been created, all calls that match the entered pattern will now display with the new formatting options, as shown below:

```

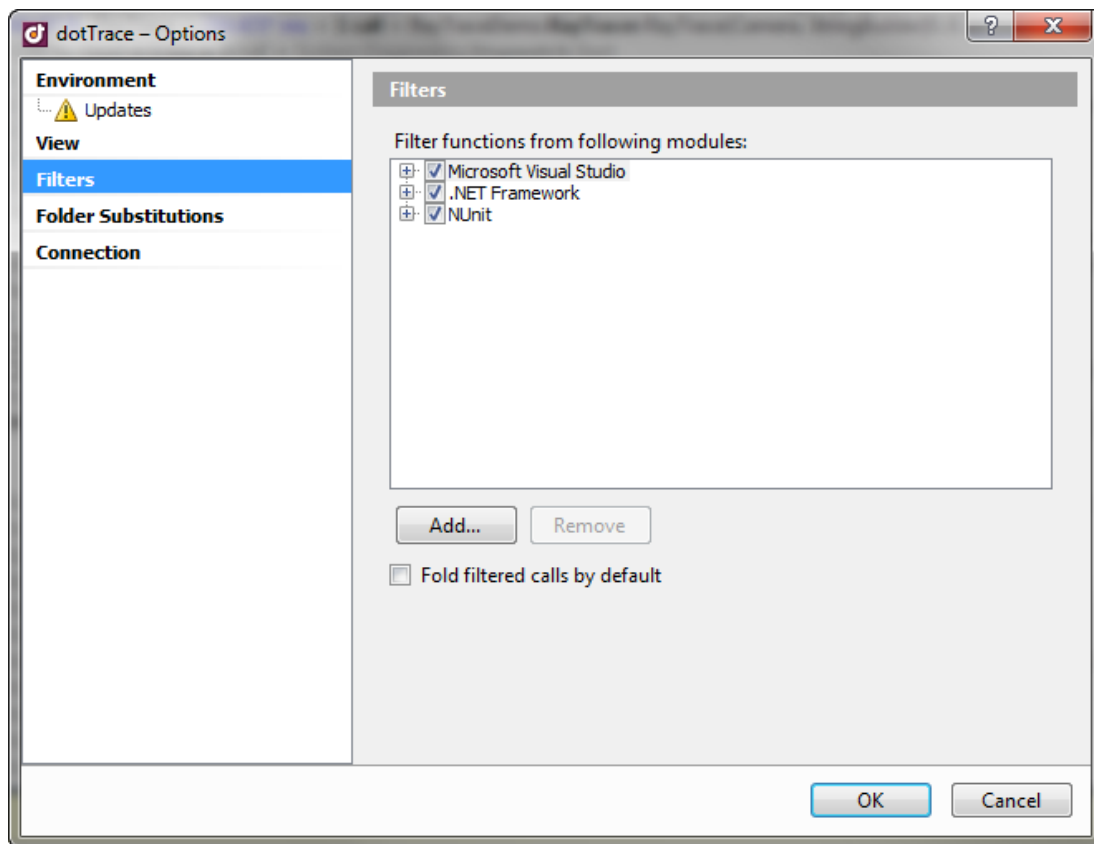
97,62 % Main • 16.901* ms • 1 call • ConsoleApplication8.Program.Main(String[0..])
97,61 % SomeMethod • 16.900* ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
80,86 % SomeMethod • 14.001 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
16,60 % Sleep • 2.873* ms • 2 calls • System.Threading.Thread.Sleep(Int32)
0,15 % WriteLine • 26 ms • 2 calls • System.Console.WriteLine(String)
0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor
2,09 % ResolvePolicy • 362 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionS
0,15 % CreateSecurityIdentity • 26 ms • 3 calls • System.Reflection.Assembly.CreateSecurityIdentity(Assembly, Si
0,06 % SetupDomain • 10 ms • 1 call • System.AppDomain.SetupDomain(Boolean, String, String)

```

Filters

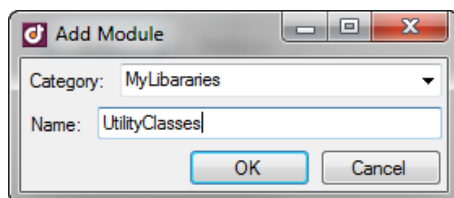
When profiling, it sometimes helps to get rid of some of the noise (third-party libraries or the BCL) and focus exclusively on our application calls. By default, dotTrace Performance provides some noise reduction for us by greying out calls to the BCL. We can not only change this setting but also add additional libraries that we want to filter out. All this can be set via the **Filters** entry under **Options**

1. Click on **View | Options** to open up the **Options** dialog and select **Filters**





By default, dotTrace Performance has Microsoft Visual Studio, .NET Framework and NUnit modules filtered out.





2. Click on the **Add** button to create a new category of Modules.
3. Give the Category a name, for example MyLibraryClass.
4. Provide the name of the namespace that you want filtered out. If for instance MyLibraryClass has a namespace UtilityClasses, we would add UtilityClasses as the Name





5. Hit the **OK** button to save the changes.

We can add as many namespaces as you want, under the same category or different categories. When we do not want the functions filtered, we can simply remove the checkbox, without having to delete them.

All filtered functions will now appear as grayed out. However, we can still access them by expanding them. This is done using the  and  icons that appear on the call line, as shown below:

  7.49% XmlSerializer.ctor • 778 ms • 761 calls • System.Xml.Serialization.XmlSerializer..ctor(Type)
  1.98% InitializeWithStatus • 205 ms • 6 calls • System.Diagnostics.Switch.InitializeWithStatus

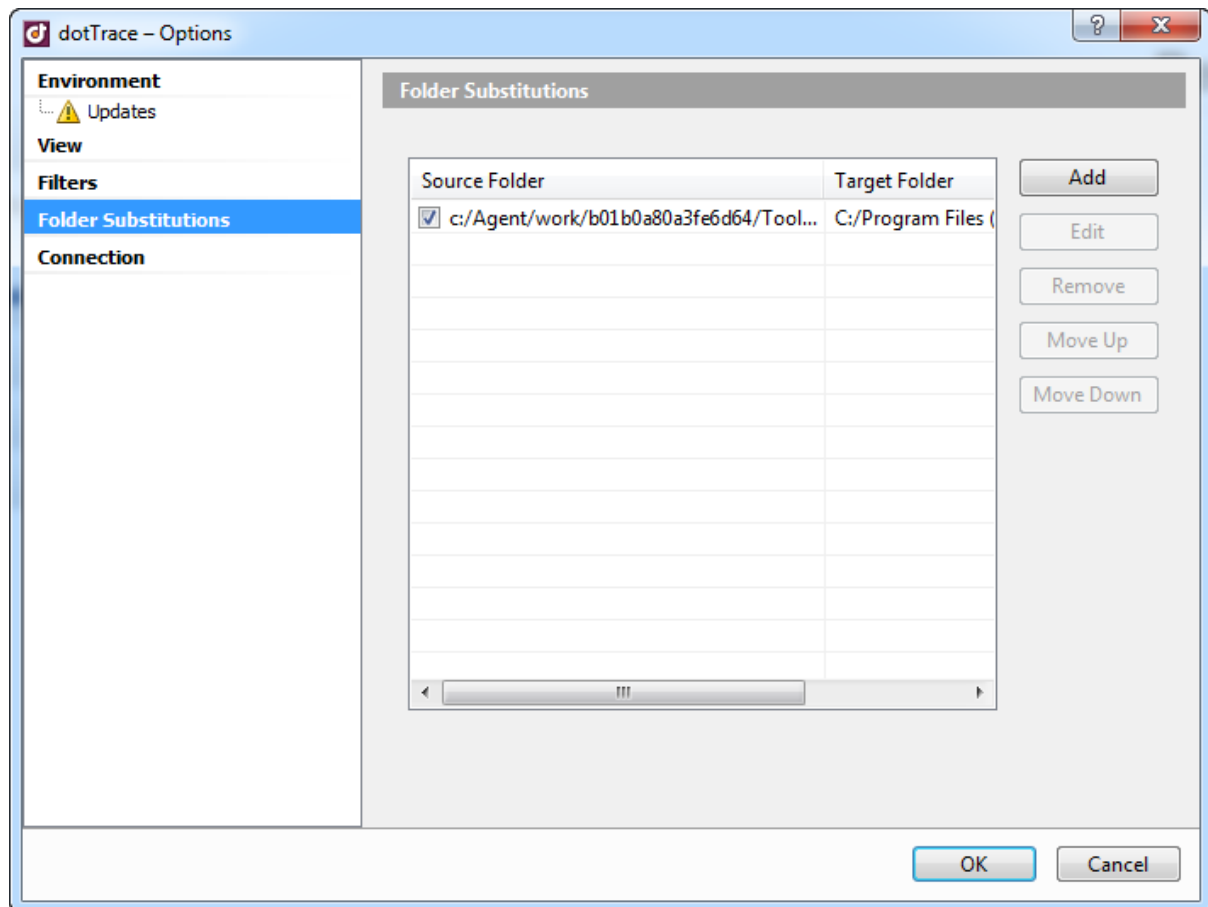
Click to Expand

  81,60 % SomeMethod • 2.026 ms • 1 call • ClassLibrary1.Class1.SomeMethod
0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor

Click to Shrink

Folder Substitutions

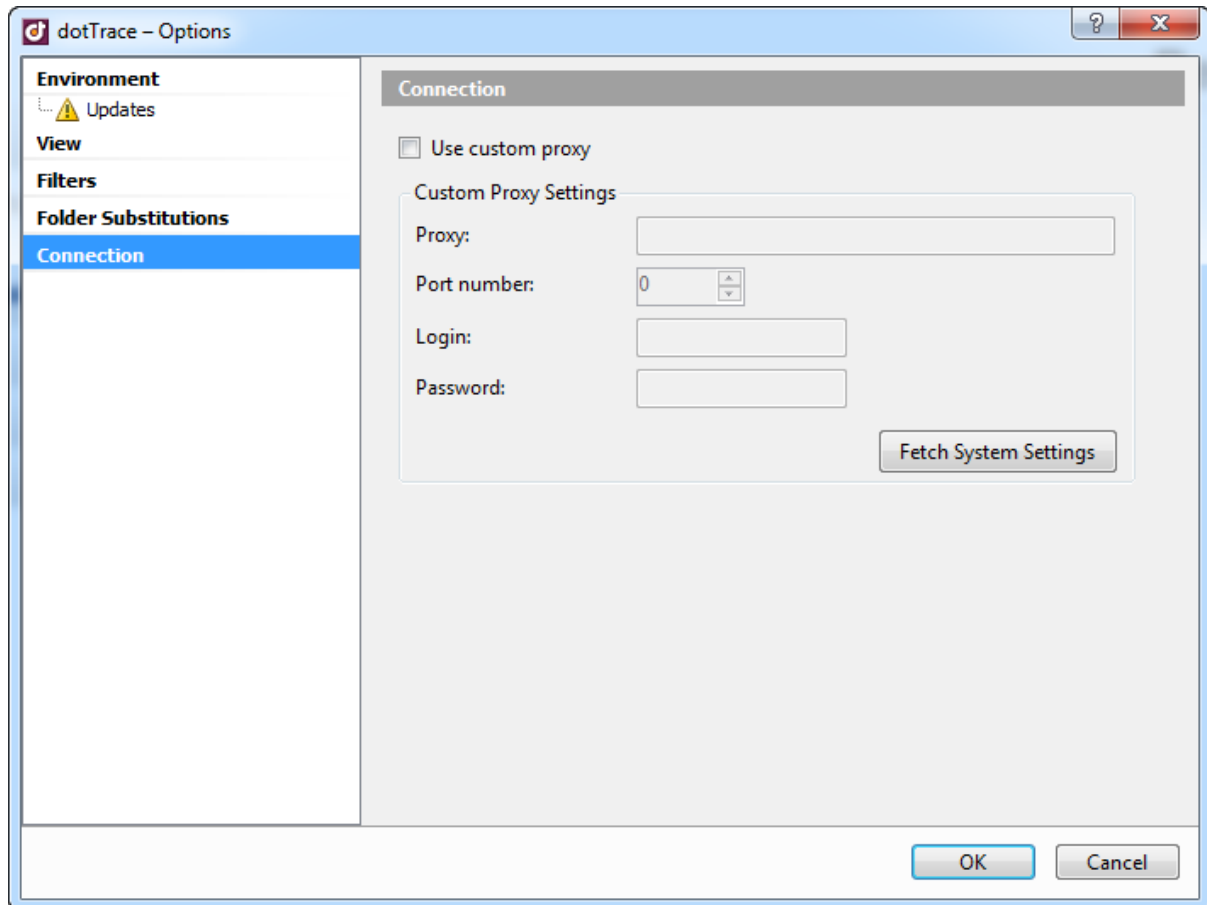
The folder substitutions allow us to change the paths to the source in PDB files when these are not the default folders. For instance, when trying to run a performance analysis on an application that has been built on a Build Agent (different machine), we can point to the new locations using Folder Substitutions.



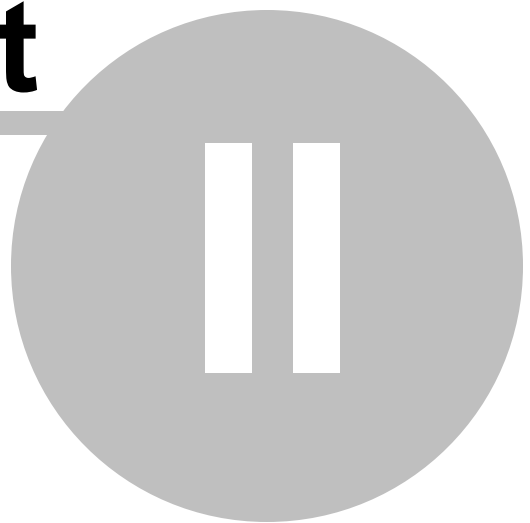
Connection

When profiling Web Applications or other types of applications that require a connection, we can optionally provide Proxy settings if we are using a Proxy to connect to the application. This can be done via the **Connection** entry under **Options**

As with other proxy configuration, we need to provide the IP or hostname of the Proxy Server, optionally the Port Number as well as authentication information if applicable. dotCover Performance can also try and detect these settings automatically by clicking on the **Fetch System Settings**



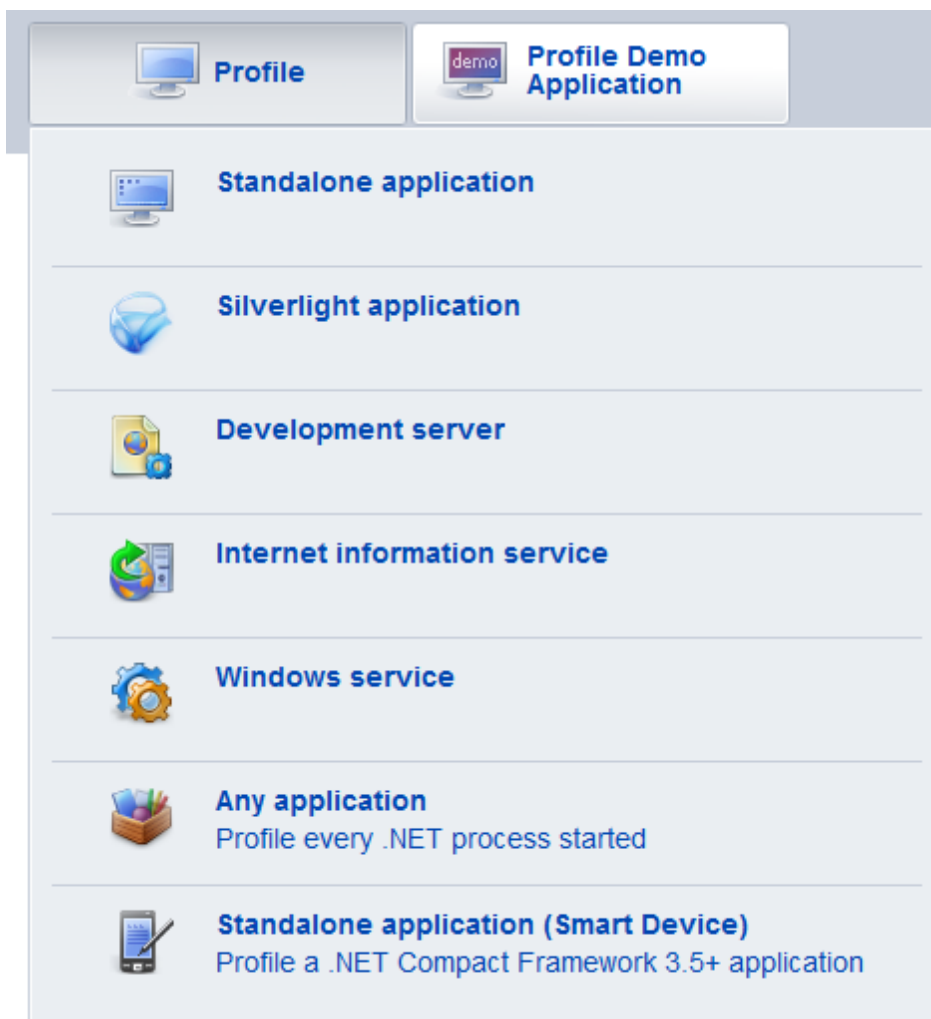
Part



2 Profiling Applications

There are two ways by which to profile applications using dotTrace Performance. We can either profile an application from within Visual Studio or run dotTrace Performance as standalone.

Profiling Applications in Standalone Mode



When running in Standalone mode, we need to provide information about the type of application we need to profile and corresponding parameters. In this section we can see all the different types of applications dotTrace Performance supports and how to set each of them up.

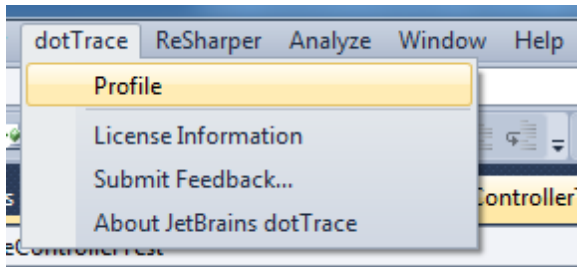
Click on one of the sections for more information.

- Standalone Application
- Silverlight Application
- Web Application using WebDev Server
- Web Application using Internet Information Service
- Windows Service
- .NET Process

- Smart Device

Profiling Applications from within Visual Studio

When profiling applications from within Visual Studio, dotTrace Performance will automatically detect the type of application we are running and configure most of the properties required for us. In order to profile, we first need to have the solution open, and then select **Profile** from the **dotTrace** menu



Clicking **Profile** will give way to a configuration screen where most of the parameters should be defined. We can adjust the Profiling Options and hit **Run**. For information on specific application type parameters, see the corresponding section.

- Standalone Application
- Silverlight Application
- Web Application using WebDev Server
- Web Application using Internet Information Service
- Windows Service
- .NET Process
- Smart Device

2.1 Types of Applications

dotTrace Performance can profile various types of applications. Depending on the type of application, different settings need to be provided. Click on one of the sections for more information.

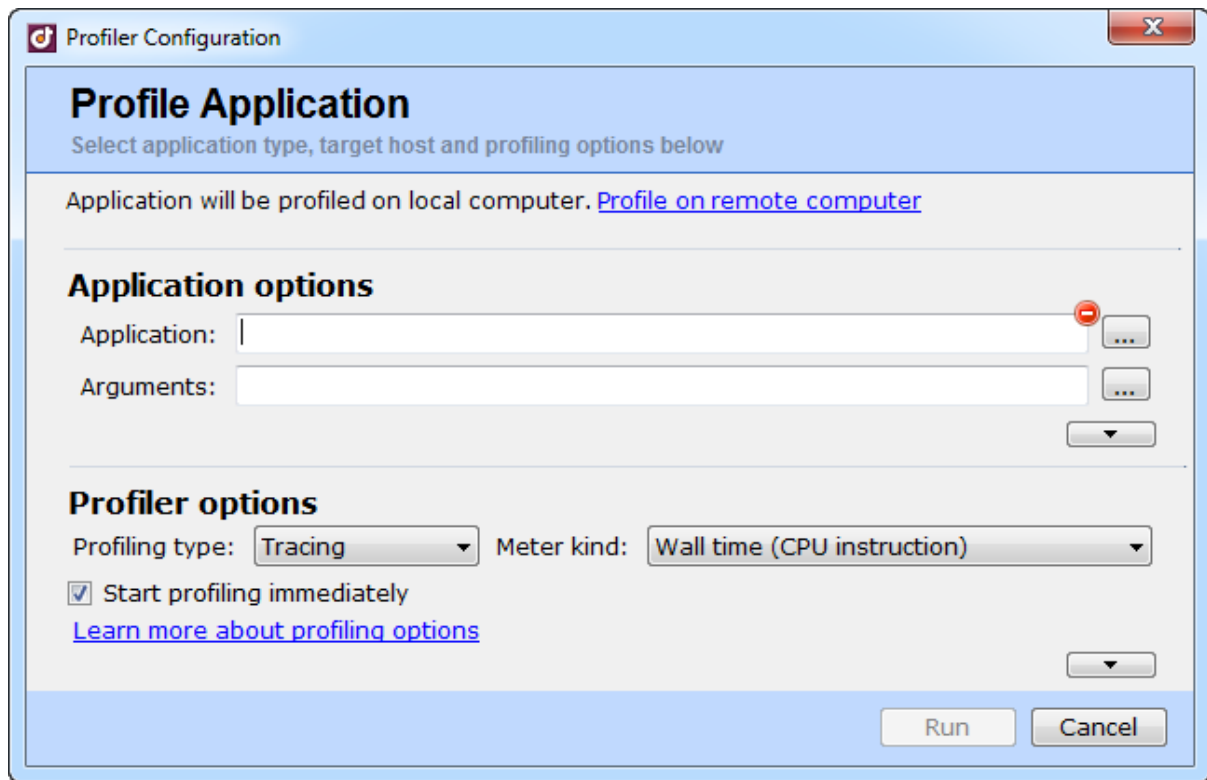
- Standalone Application
- Silverlight Application
- Web Application using WebDev Server
- Web Application using Internet Information Service
- Windows Service
- .NET Process
- Smart Device

2.1.1 Standalone Applications

Standalone applications include Console Application, Windows Forms or Windows Presentation Foundation applications. In order to profile a Standalone Application, we need to provide the following parameters:

- Application: Complete path to the executable
- Arguments: Optional Arguments passed to the executable

Once these options are provided, we can click **Run** to have the application start and the dotTrace Controller appear. For advanced profiling options, please see Profiling Options

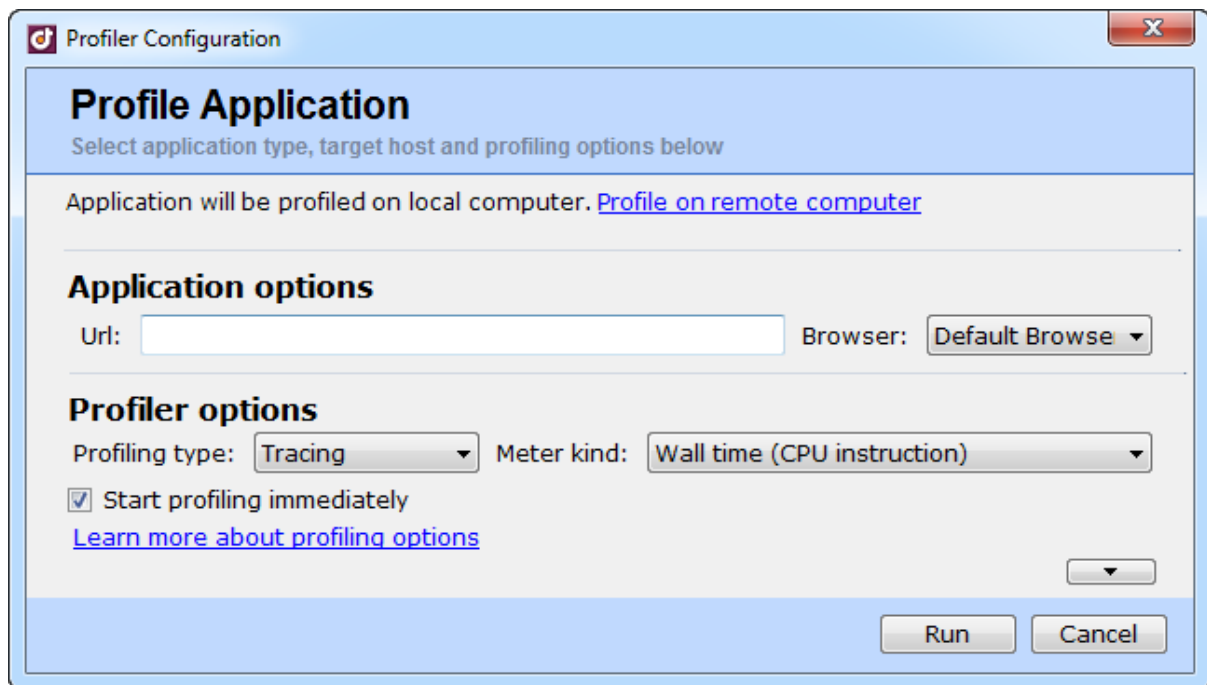


2.1.2 Silverlight Applications

Silverlight 4 provides a profiling API and as such dotTrace Performance can take advantage of this to perform profiling. In order to profile a Silverlight Application, we need to provide the following parameters:

- Url: Url to invoke the Silverlight Application. If no Url is provide, dotTrace Performance will simply launch the browser.
- Browser: Which Browser we wish to use to run the application to profile. By default the Default system browser is selected.

Once these options are provided, we can click **Run** to have the application start and the dotTrace Controller appear. For advanced profiling options, please see Profiling Options



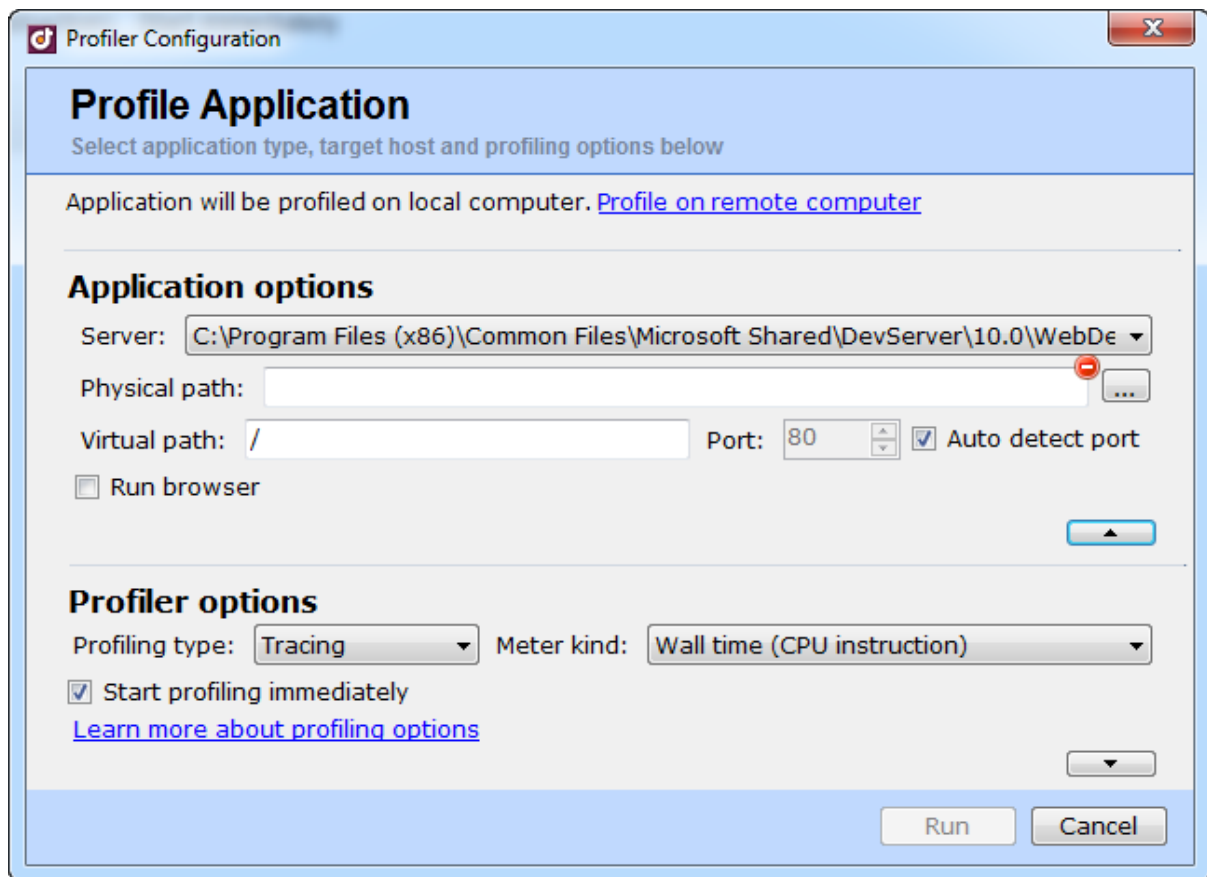
2.1.3 Web Application using WebDev Server

Web Applications can be profiled whether they are using the WebDev Server, which is usually during development, or running on Internet Information Services. In order to profile a Web Application using WebDev Server, we need to provide the following parameters:

- Server: Complete path to the WebDev Server. dotTrace Performance usually can detect this by default.
- Physical Path: Complete path to the root folder of the Web Application
- Run Browser: Optionally we can specify whether we want dotTrace Performance to launch the browser for us. If we choose to, we can provide additional information:
 - Virtual Path: The virtual path to start the application
 - Port: The port used by the server. Default is 80
 - AutoDetect Port: Will automatically try and detect the port

Once these options are provided, we can click **Run** to have the application start and the dotTrace Controller appear. For advanced profiling options, please see Profiling Options

Note: dotTrace cannot automatically shutdown the server when existing a profiling session. As such, often when trying to run a second profiling session, we might receive an error. In order to solve the problem, we need to shut down the WebDev server by clicking Stop or exiting it. The application icon normally sits in the System Tray.

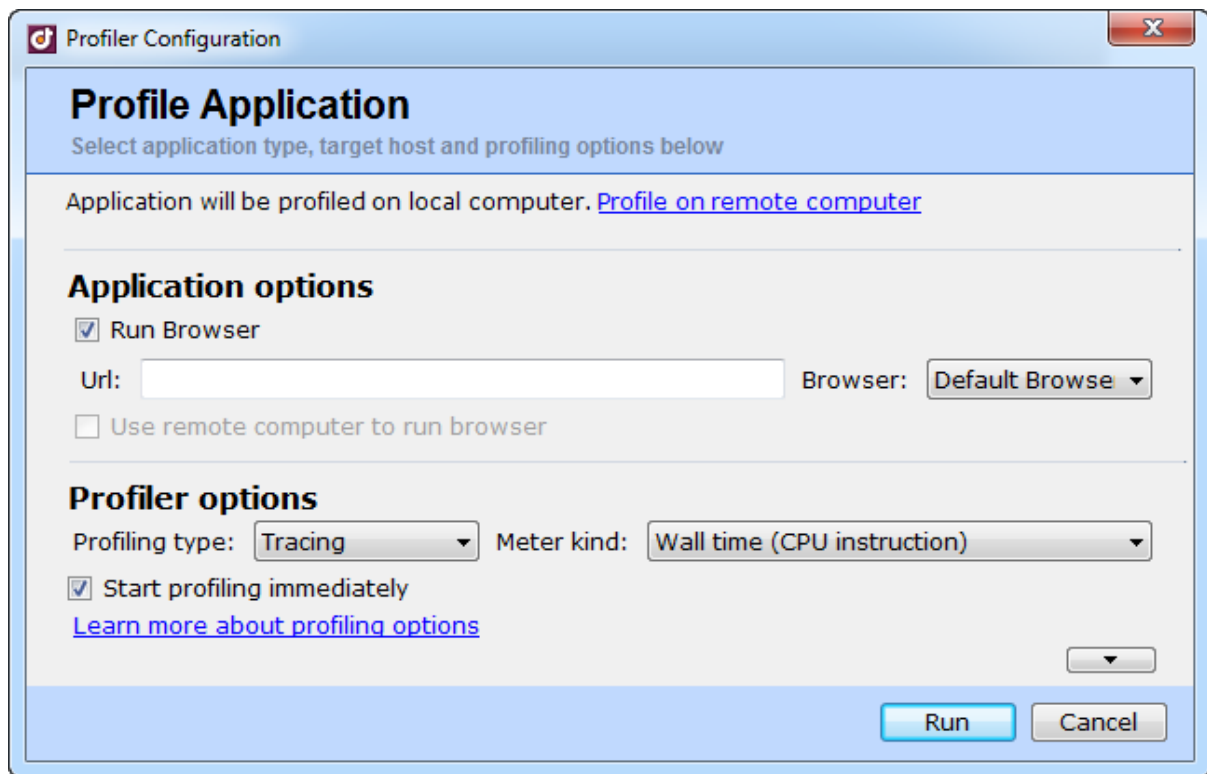


2.1.4 Web Application using Internet Information Service

Web Applications can be profiled whether they are using the WebDev Server, which is usually during development, or running on Internet Information Services. It is important to note that when profiling applications using Internet Information Services, dotTrace Performance needs to run under Administrative Privileges (In Windows Vista or 7, right-click and select Run as administrator). In order to profile a Web Application using Internet Information Services, we can just run the profiler and it will attach itself to Internet Information Services. We can optionally select:

- Run Browser: Selecting this option we provide additional properties:
 - Url: The Url to invoke the application
 - Browser: The Browser to use. By default it selects the system Default Browser.

Once these options are provided, we can click **Run** to have the application start and the dotTrace Controller appear. For advanced profiling options, please see Profiling Options

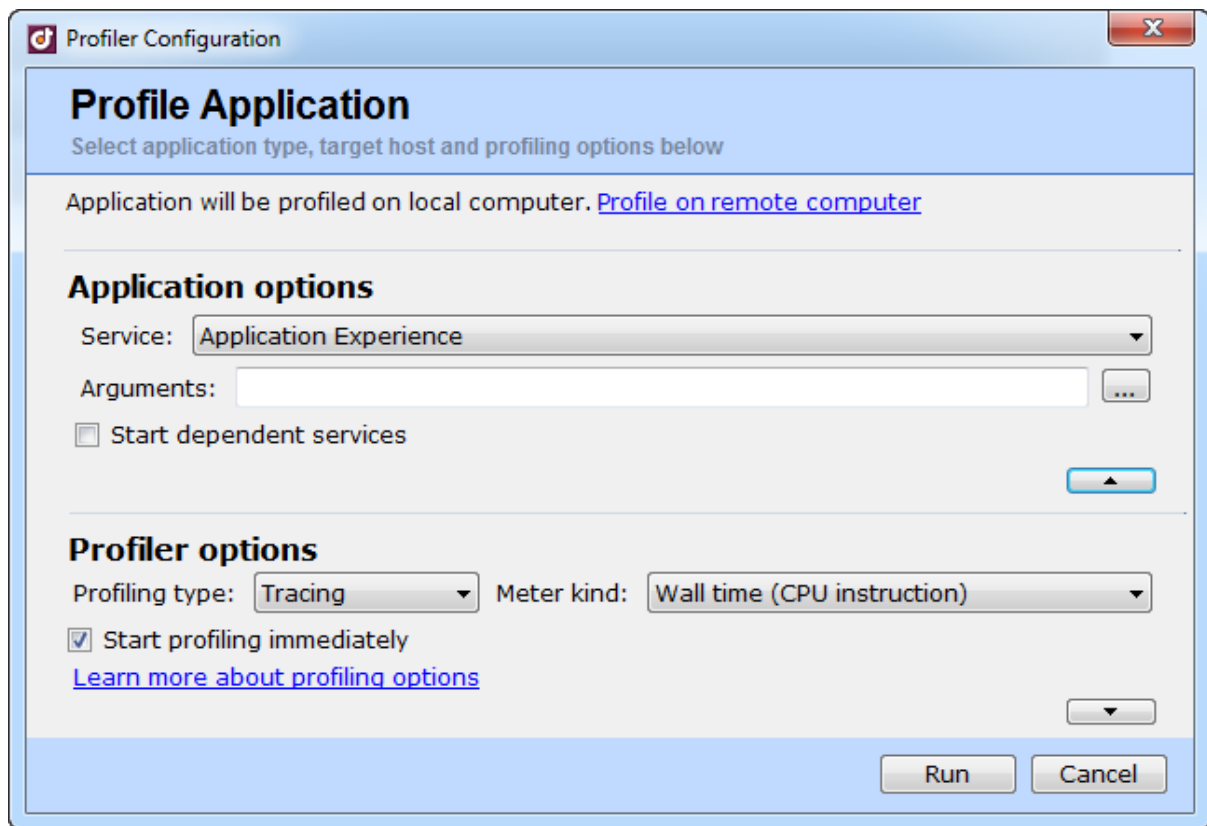


2.1.5 Windows Services

Any Managed Windows Service can be profiled using dotTrace Performance. In order to profile a Windows Service, we need to provide the following parameters:

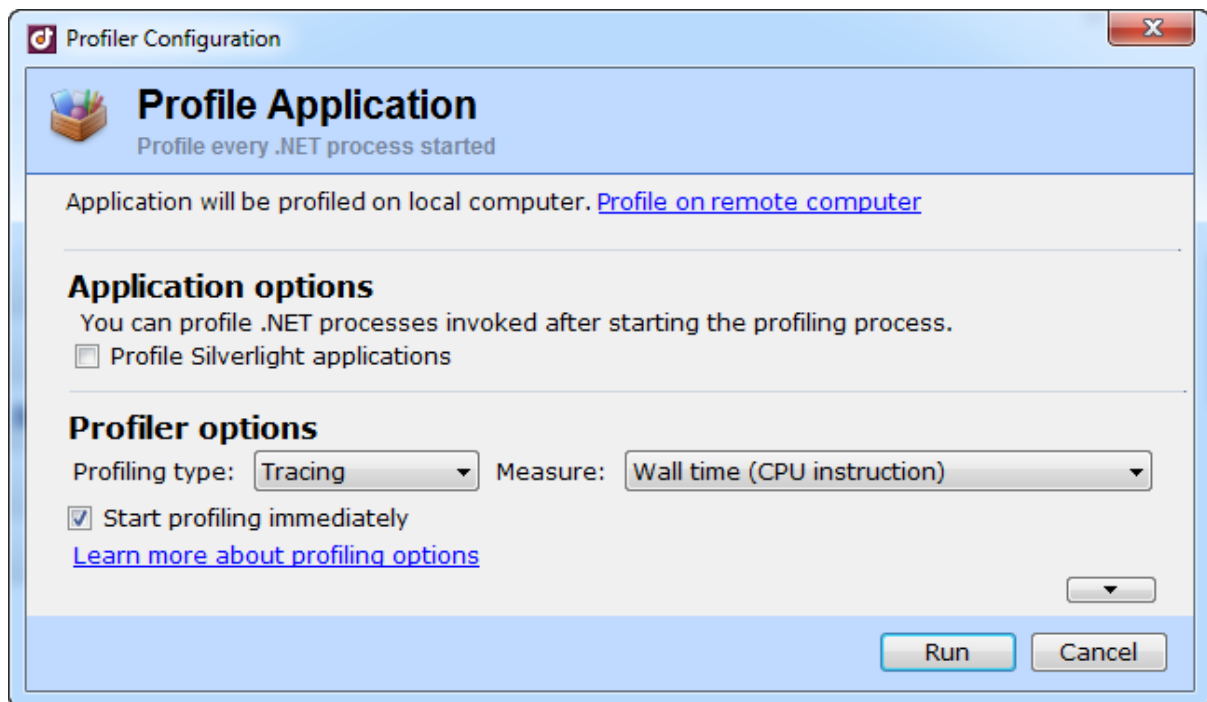
- Service: Name of the service to profile. dotTrace Performance scans the installed services automatically. You need to have the service installed for it to appear in the list.
- Arguments: Optional Arguments passed to the service
- Start Dependent Services: If the service requires other services to be started, by checking this checkbox dotTrace Performance will start them.

Once these options are provided, we can click **Run** to have the application start and the dotTrace Controller appear. For advanced profiling options, please see Profiling Options

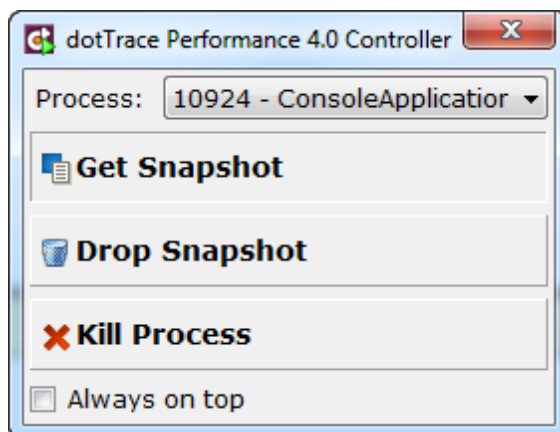


2.1.6 .NET Process

Profiling a .NET Process allows us to profile any managed application*. In order to profile a .NET Process, all we need to do is select the .NET Process from the Profile menu and indicate the profiling options, that is the Profiling Type and Measurement. We can optionally indicate to dotTrace Performance to include Silverlight applications.



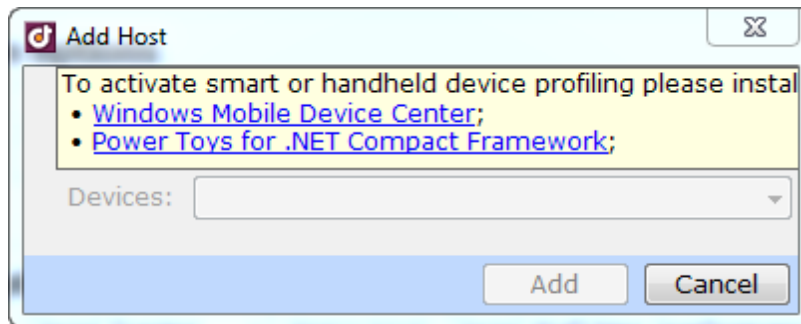
We then click Run and dotTrace launches the Controller. From this point on, dotTrace is ready to hook into any managed application that is launched. If more than one application gets launched, dotTrace will allow us to select the snapshot we are interested in by means of a DropDown in the Controller.



* Any Application that is launched AFTER launching dotTrace Performance Profiling, that is hitting Run

2.1.7 Smart Devices (Standalone Application)

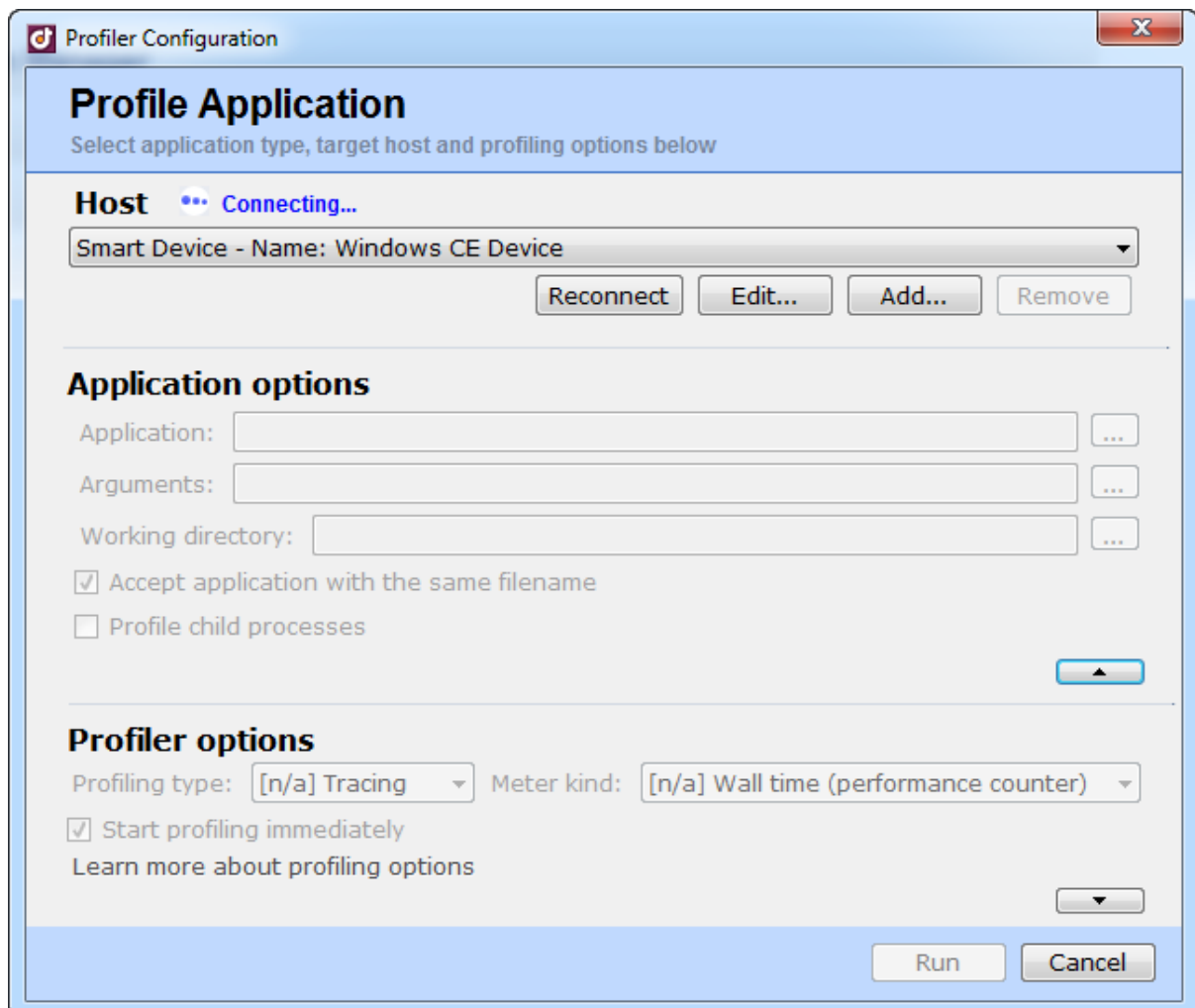
Smart Devices can be profiled using dotTrace Performance. There are certain software packages that need to be installed to allow for dotTrace Performance to work, in particular these are the Windows Mobile Device Center and the Power Toys for .NET Compact Framework. These packages are usually already installed on systems dedicated to device development. dotTrace Performance warns of these requirements when Selecting a Device, as shown below:



Once the pre-requisites are installed, in order to profile devices, we need to provide dotTrace Performance with the following parameters:

- Host: The host device we are going to profile. This is selected via a dropdown list

Once the host is provided, dotTrace Performance will then attempt to connect to the device:



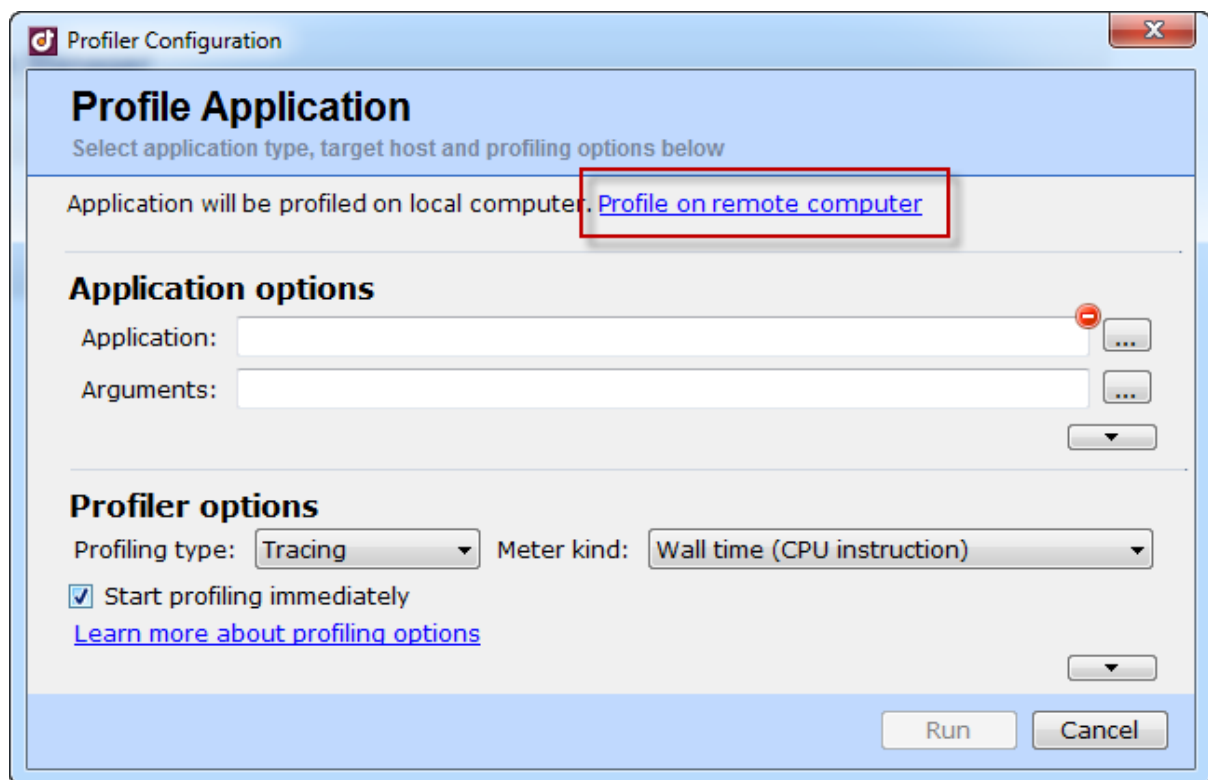
Once a successful connection has been established, we then need to provide dotTrace with:

- Application: Complete path to the application we want to profile on the device
- Arguments: Optional arguments to pass to the application
- Working directory: The working directory used on the device

2.2 Remote Profiling

dotTrace Performance provides us with a powerful feature of being able to run profiling on a different machine from where dotTrace Performance is installed. Often we find that performance issues cannot be replicated accurately on a development machine or environment and sometimes it is necessary to run a profiling session on a production machine.

We can do this easily with dotTrace without the need to install it locally on the machine. In order to perform a remote profile, all the steps are the same as with a local profile, that is, we need to select the type of application and provide the required parameters. The additional step required is to specify information about the remote machine, which is done by clicking on the **Profile on Remote Computer** link present on all Configuration screens:

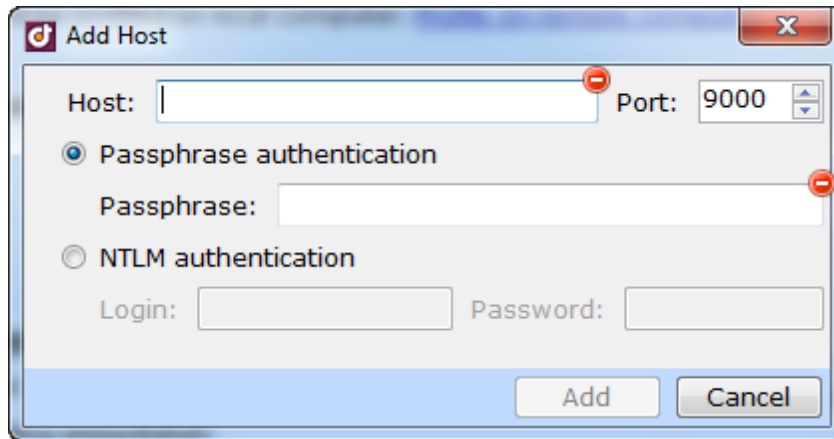


This will provide us with an additional screen where we can define the following parameters:

- Host: The IP or hostname of the machine

- Port: The Port dotTrace Performance will connect to. By default it is port 9000
- Passphrase authentication: dotTrace Performance allows for two types of authentication, custom dotTrace which is a passphrase or
- NTLM authentication: A valid Windows account on the remote machine

On first connection, dotTrace Performance will automatically install a remote tool to allow for profiling to take place. It is important to have the selected port open (firewall properly configured).

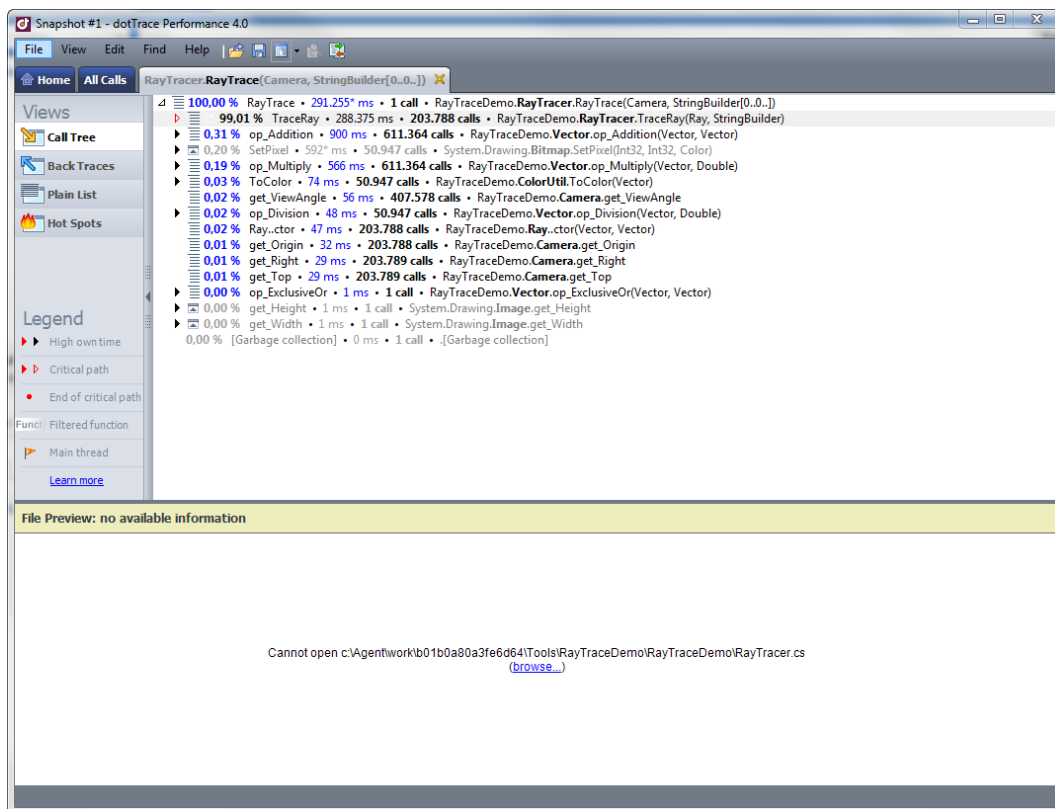


2.3 Working with Source Code

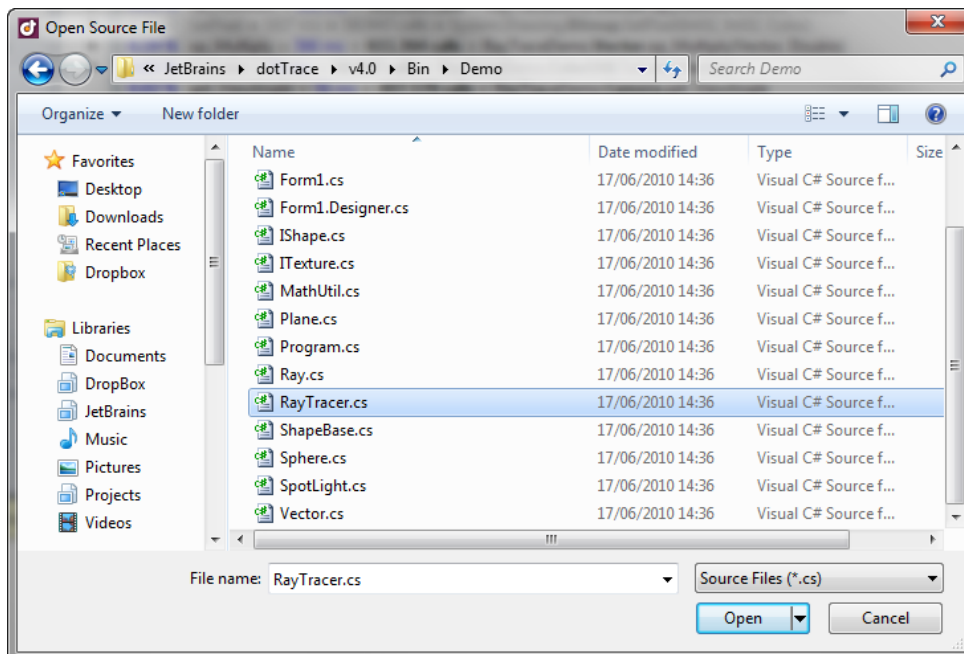
One of dotTrace Performance greatest features is its integration with source code. dotTrace Performance allows us to view specific function calls in the context of the original source code, whenever this is available.

Working with source code depends on how dotTrace Performance was launched. If launching via Visual Studio Profiling Option source code is already available and configured for browsing, so usually no additional steps are required. If using dotTrace Performance by launching it externally we need to follow a series of steps in order to have the source code available:

1. Click on the function call for which you want to examine the source code. The File Preview window below will be display a message indicating that it cannot open the specific file

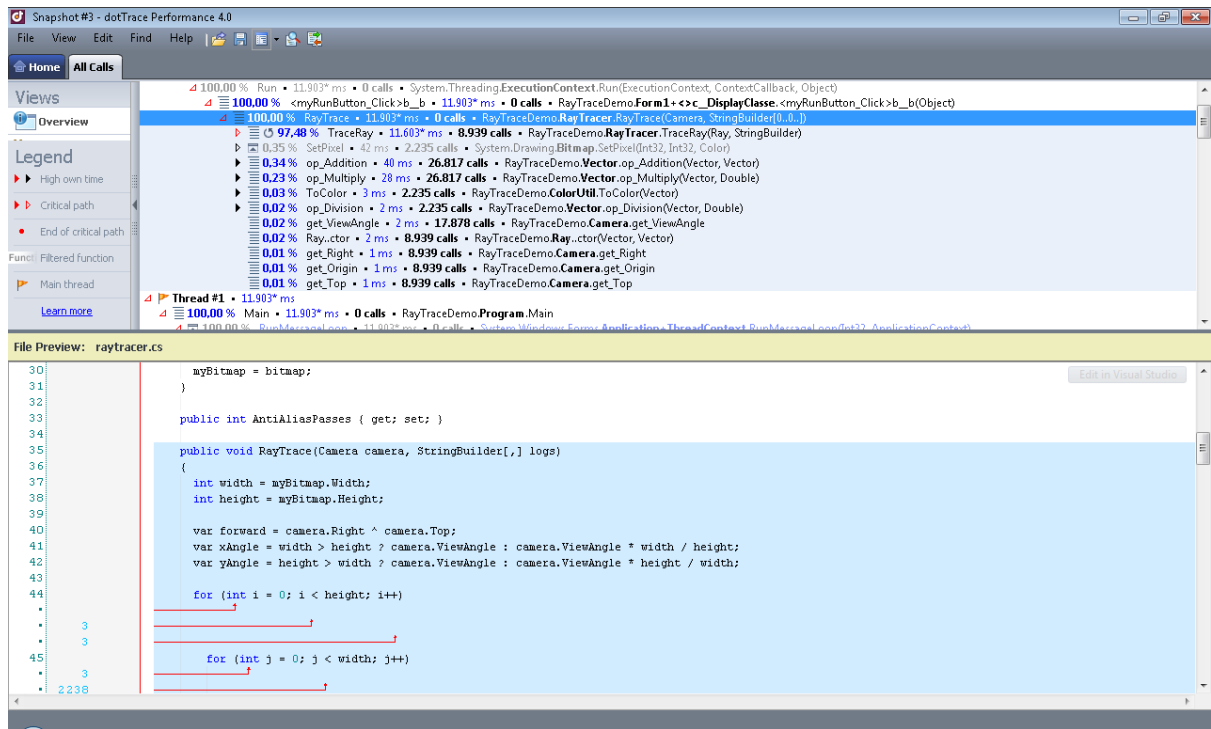


- Click on the **browse..** button in the File Preview window to invoke the **Open File** dialog box



- Locate the file in the dialog and click **Open**

The File Preview window should now display the source code, as shown below

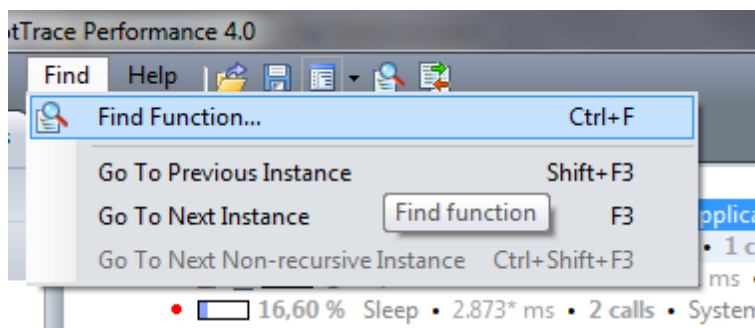


2.4 Locating Functions

When we need to profile a specific function, instead of having to navigate the call tree to find the find the call, we can use dotTrace Performance built-in function lookup to locate it immediately.

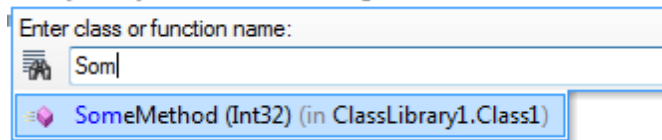
In order to locate a function:

1. Click on **Find | Find Function** menu to invoke the Find dialog box

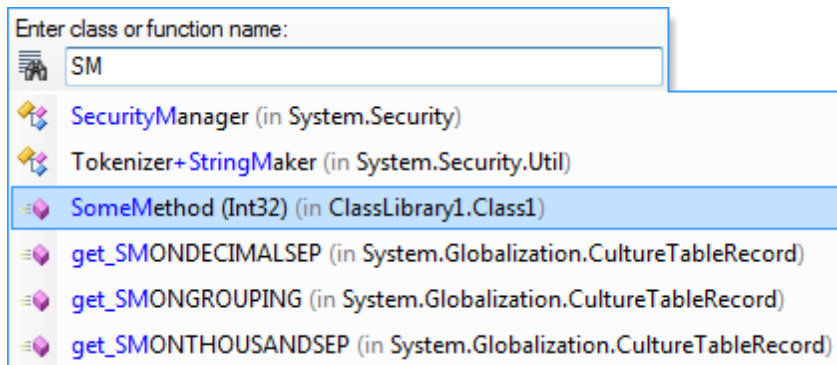


2. Enter the name of the function. The dialog box support CamelCase as well as wildcard patterns (for

those familiar with ReSharper, this is the same infrastructure)

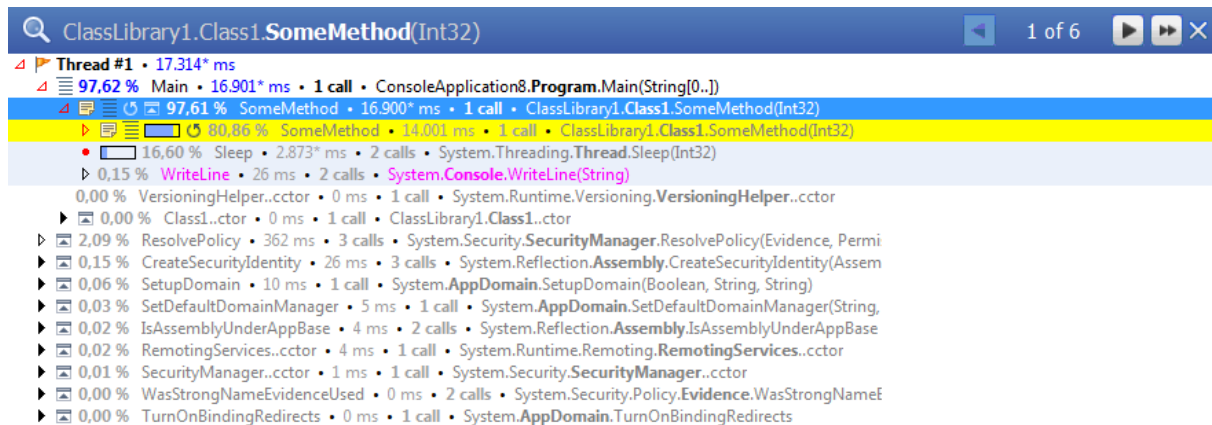


Using 'Som' to locate SomeMethod



Using CamelCase (SM) to locate same function

Once we hit the Enter key, dotTrace Performance will locate the call for us anywhere it might appear in the call stack. We can now navigate to all instances uses F3 and Shift+F3, which go to the Next and Previous instances respectively. We can also use the Navigation buttons that appear in the Search Result to accomplish the same effects, as shown below:

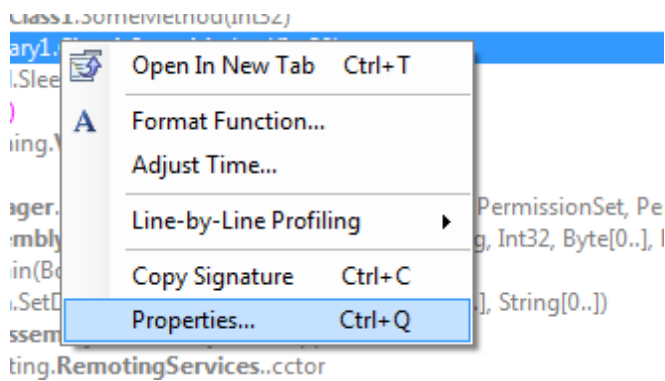


2.5 Annotating Functions

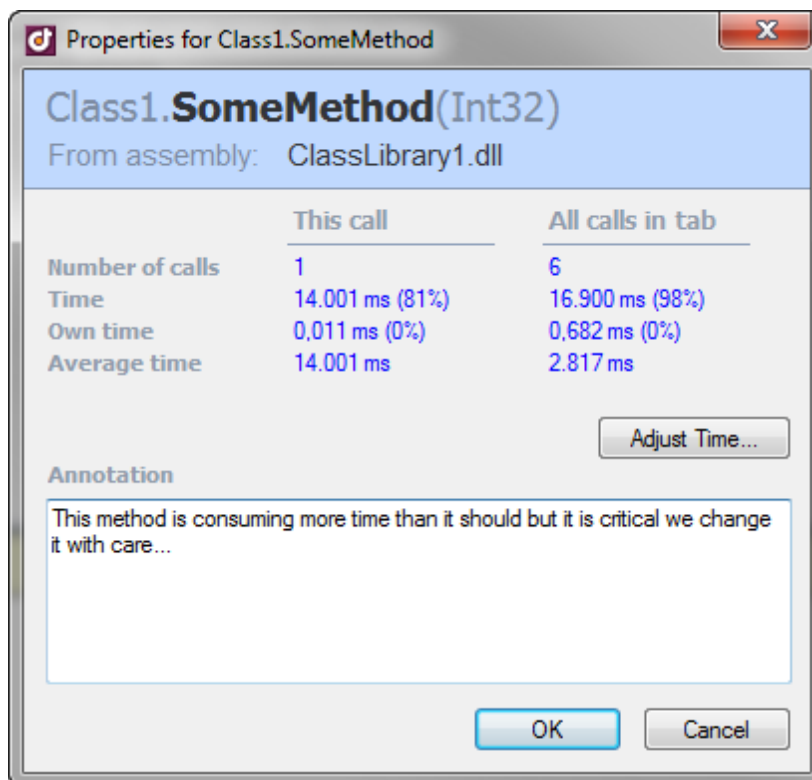
Sometimes it is necessary to write comments or take down notes around specific functions we are profiling so that later on we can recall all the information gathered or analysed. dotTrace Performance allows us to do this by offering us Annotation.

In order to annotate a function:


1. Right-click on the function that requires annotating to bring up the **Properties** Context Menu. We can also press Ctrl-Q to invoke this.



2. In the Properties dialog box, we can now type any annotations, comments or notes we like about the specific function



3. Hit **OK** to save the changes

We can now observe that the function call has a new *PostIt* style icon  next to it, indicating that the function has an associated annotation, as shown below

```

Thread #1 • 17.314* ms
  97,62 % Main • 16.901* ms • 1 call • ConsoleApplication8.Program.Main(String[0..])
    97,61 % SomeMethod • 16.900* ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
      80,86 % SomeMethod • 14.001 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
        16,60 % Sleep • 2.873* ms • 2 calls • System.Threading.Thread.Sleep(Int32)
          0,15 % WriteLine • 26 ms • 2 calls • System.Console.WriteLine(String)
            0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
              0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor

```

2.6 Adjusting Call Times

When locating bottlenecks, it is often useful to see how removing it or somehow minimizing its impact would reflect on the overall performance analysis. Normally this involves solving the particular issue, which could involve re-design of an algorithm or changes to the program structure. This is not an immediate process and among other things involves a recompile of the application. dotTrace Performance allows us to simulate these changes and see its impact on the overall performance by permitting us to adjust call times.

1. In order to do so, we first need to locate the specific function that we want to adjust the time.

```

0,15 % WriteLine • 26 ms • 2 calls • System.Console.WriteLine(String)
  0,14 % get_Out • 24 ms • 2 calls • System.Console.get_Out
    0,01 % WriteLine • 2 ms • 2 calls • System.IO.TextWriter+SyncTextWriter.WriteLine(String)
      0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
        0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor
  2,09 % ResolvePolicy • 362 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, Permiss
    2,07 % ResolvePolicy • 359 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, Permiss
      0,00 % AssertAllPossible • 0 ms • 3 calls • System.Security.CodeAccessPermission.AssertAllPossible
        0,00 % GetSpecialFlags • 0 ms • 3 calls • System.Security.SecurityManager.GetSpecialFlags(PermissionSet, PermissionSet)

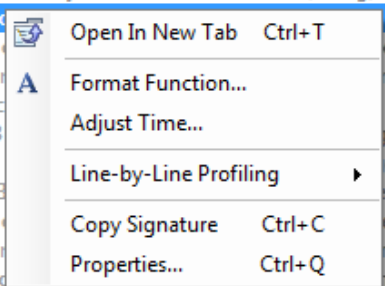
```

2. Right-click on the function to select **Adjust Time** from the Context Menu

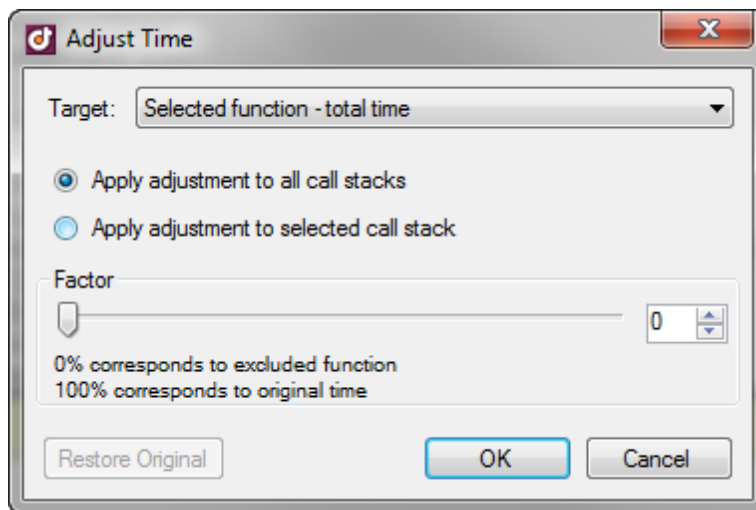
```

0,15 % WriteLine • 26 ms • 2 calls • System.Console.WriteLine(String)
  0,14 % get_Out • 24 ms • 2 calls • System.Console.get_Out
    0,01 % WriteLine • 2 ms • 2 calls • System.IO.TextWriter+SyncTextWriter.WriteLine(String)
      0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
        0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor
  2,09 % ResolvePolicy • 362 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, Permiss
    2,07 % ResolvePolicy • 359 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, Permiss
      0,00 % AssertAllPossible • 0 ms • 3 calls • System.Security.CodeAccessPermission.AssertAllPossible
        0,00 % GetSpecialFlags • 0 ms • 3 calls • System.Security.SecurityManager.GetSpecialFlags(PermissionSet, PermissionSet)
  0,15 % CreateSecurityIdentity • 26 ms • 1 call • System.Security.Principal.SecurityIdentifier.CreateSecurityIdentity(Assembly, Stri
    0,06 % SetupDomain • 10 ms • 1 call • System.Security.Principal.SecurityIdentifier.SetupDomain(Assembly, String, String)

```



3. The **Adjust Time** dialog box appear



The dialog box allows us to set the function's own time, the total time or all functions of that specific class, by specifying the Target. In this case we are going to set the function's total time.

- Next we can select either **Apply adjustment to all call stacks** or **Apply adjustment to selected call stack**. This refers to whether the reduction is going to be applied to all functions across all call stacks or just the selected one. In this case we choose just the selected stack.
- Set the **Factor** to 25%. This indicates what percentage of the time we are going to reduce it by.
- Hit the **OK** button.

The call stack should now reflect this adjustment, as shown below:

```
▲ 0,05 % WriteLine • 8 ms (was 26) • 2 calls • System.Console.WriteLine(String)
▶ 0,14 % get_Out • 24 ms • 2 calls • System.Console.get_Out
  ▶ 0,01 % WriteLine • 2 ms • 2 calls • System.IO.TextWriter+SyncTextWriter.WriteLine(String)
0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
▶ 0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor
▶ 2,09 % ResolvePolicy • 362 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, Pe
▶ 2,07 % ResolvePolicy • 359 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, Pe
▶ 0,00 % AssertAllPossible • 0 ms • 3 calls • System.Security.CodeAccessPermission.AssertAllPossible
```

The total call time has gone from 26ms to 8ms. If we do the corresponding calculations we see how this number has come about. We originally had 26ms composed of 2 parts, one of **get_Out** of 24ms and another of **WriteLine** of 2ms. By reducing the **get_Out** call to 25% of its original time, it would now be 6ms, which added to **WriteLine** 2ms, gives us a total of 8ms.

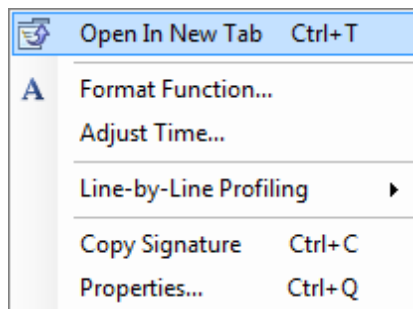
- To restore the original time, we can call the **Adjust Time** function again and click on the **Restore Original** button.

2.7 Working with Tabs

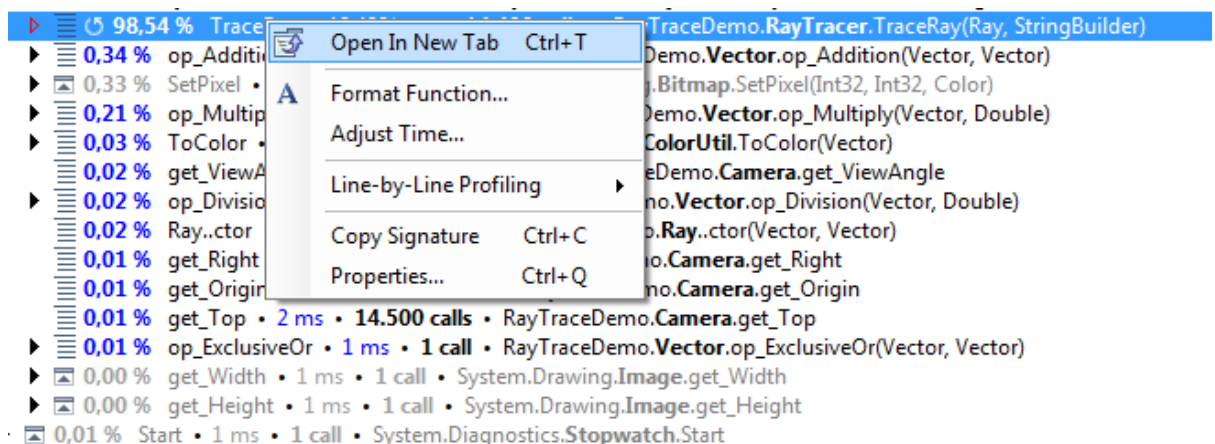
One of dotTrace Performance features is to allow comparison of multiple snapshots. In order to make this easy, as well as making analysis of call stacks simpler, dotTrace Performance allows snapshots to be displayed in separate tabs. At any point we can move an entire function and the call stack below it to a new tab for better focused analysis or future comparisons.

In order to use tabs:

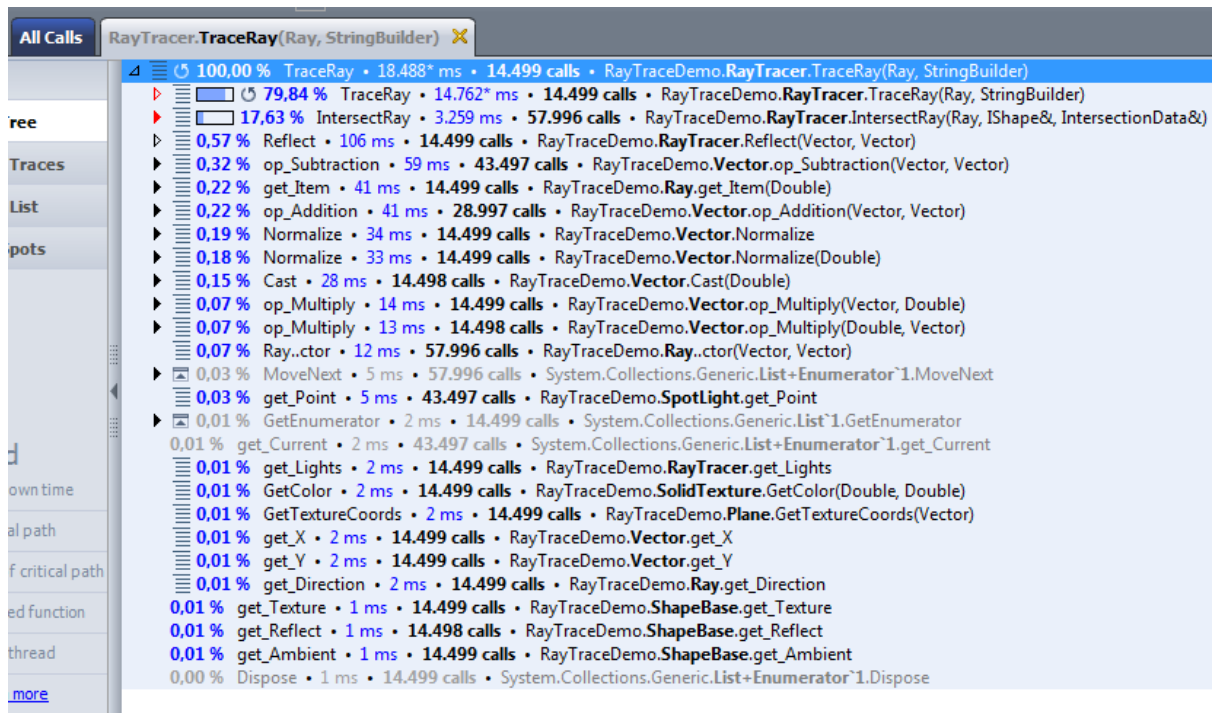
1. Right-click on the function method you want to move to a new Tab to invoke the **Open in New Tab** option of the Context Menu. You can alternatively use **Ctrl+T**.



2. If the function is called from multiple places, dotTrace Performance will prompt us to see if we want to merge all the calls into the same new tab we are creating. If this is the case, and based on what we need the information for, we need to reply with Yes in which case it will merge the calls, or No, in which case it will only display that specific call.



Once we confirm the operation, a new Tab will be created as shown below:





It is important to notice that in the new Tab, the function call now represents 100% of the entire call stack, that is, the percentage is always relative to the entire stack represented.

2.8 Flattening Recursive Calls

Recursive calls in our code can be represented in two ways with dotTrace Performace:

- Unfolded: Each call is represented separately, with it's own time and overall percentage
- Folded: All recursive calls are folded into one call that displays the total time and overall percentage.

In order to switch between these, we can use the unfold  and fold  icons that appear next to the corresponding function call, as displayed below:

```

  97,62 % Main • 16.901* ms • 1 call • ConsoleApplication8.Program.Main(String[0..])
  97,61 % SomeMethod • 16.900* ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  80,86 % SomeMethod • 14.001 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  63,54 % SomeMethod • 11.000 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  46,21 % SomeMethod • 8.000 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  28,88 % SomeMethod • 5.000 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  17,32 % Sleep • 2.999 ms • 2 calls • System.Threading.Thread.Sleep(Int32)
  11,55 % SomeMethod • 2.000 ms • 1 call • ClassLibrary1.Class1.SomeMethod(Int32)
  11,55 % Sleep • 2.000 ms • 1 call • System.Threading.Thread.Sleep(Int32)
  0,00 % WriteLine • 0 ms • 1 call • System.Console.WriteLine(String)
  0,00 % WriteLine • 0 ms • 2 calls • System.Console.WriteLine(String)
  17,33 % Sleep • 3.000 ms • 2 calls • System.Threading.Thread.Sleep(Int32)
  0,00 % WriteLine • 0 ms • 2 calls • System.Console.WriteLine(String)
  17,32 % Sleep • 3.000 ms • 2 calls • System.Threading.Thread.Sleep(Int32)

```

Unfolded View

```

  97,61 % SomeMethod • 16.900 ms • 6 calls • ClassLibrary1.Class1.SomeMethod(Int32)
  97,45 % Sleep • 16.871* ms • 11 calls • System.Threading.Thread.Sleep(Int32)
  0,16 % WriteLine • 28 ms • 11 calls • System.Console.WriteLine(String)
  0,00 % VersioningHelper..cctor • 0 ms • 1 call • System.Runtime.Versioning.VersioningHelper..cctor
  0,00 % Class1..ctor • 0 ms • 1 call • ClassLibrary1.Class1..ctor
  2,09 % ResolvePolicy • 362 ms • 3 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, PermissionSet, PermissionSet)

```

Folded View

2.9 Interpreting Call Information

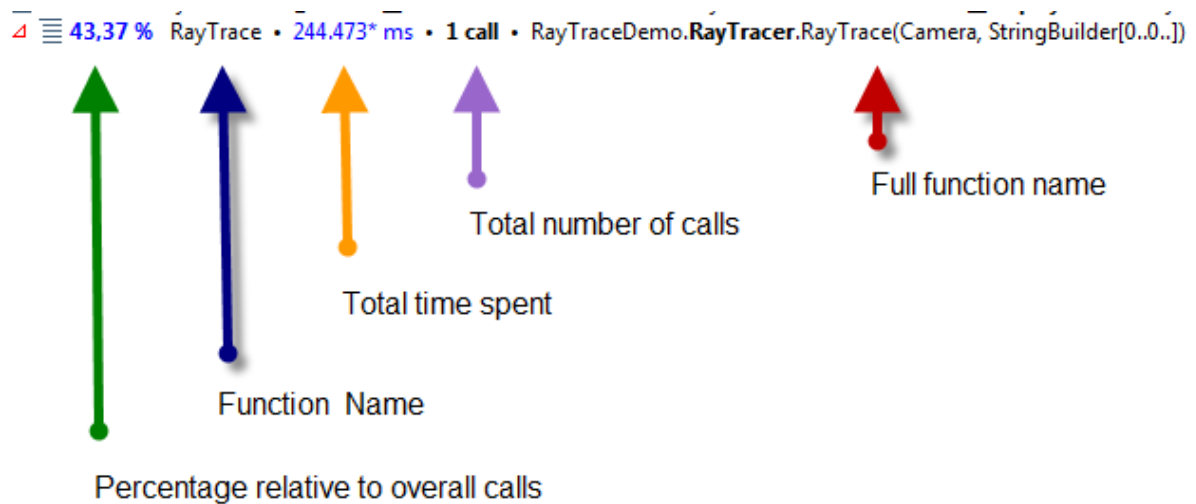
Nearly all views represent application calls using the same representation. In this section we will see how to interpret this information.

```

  43,38 % PerformWaitCallback • 244.483* ms • 1 call • System.Threading.ThreadPoolWaitCallback.PerformWaitCallback(Object)
  43,38 % PerformWaitCallbackInternal • 244.482* ms • 1 call • System.Threading.ThreadPoolWaitCallback.PerformWaitCallbackInternal(ThreadPoolWaitCallback)
  43,38 % Run • 244.482* ms • 1 call • System.Threading.ExecutionContext.Run(ExecutionContext, ContextCallback, Object)
  43,38 % <myRunButton_Click>b_b • 244.481* ms • 1 call • RayTraceDemo.Form1+<c__DisplayClass>.myRunButton_Click(b_b)(Object)
  43,37 % RayTrace • 244.473* ms • 1 call • RayTraceDemo.RayTracer.RayTrace(Camera, StringBuilder[0..])
  42,93 % TraceRay • 241.961* ms • 166.413 calls • RayTraceDemo.RayTracer.TraceRay(Ray, StringBuilder)
  0,14 % op_Addition • 766 ms • 499.238 calls • RayTraceDemo.Vector.op_Addition(Vector, Vector)
  0,10 % SetPixel • 561 ms • 41.603 calls • System.Drawing.Bitmap.SetPixel(Int32, Int32, Color)
  0,09 % op_Multiply • 498 ms • 499.239 calls • RayTraceDemo.Vector.op_Multiply(Vector, Double)
  0,01 % ToColor • 66 ms • 41.603 calls • RayTraceDemo.ColorUtil.ToColor(Vector)
  0,01 % get_ViewAngle • 43 ms • 332.828 calls • RayTraceDemo.Camera.get_ViewAngle
  0,01 % op_Division • 43 ms • 41.603 calls • RayTraceDemo.Vector.op_Division(Vector, Double)
  0,01 % Ray..ctor • 40 ms • 166.413 calls • RayTraceDemo.Ray..ctor(Vector, Vector)

```

Each line consists of the following data:



- **Percentage relative to overall calls:** Represents the percentage that this function represents in relation to the overall number of calls present. This percentage is based on the current call stack, as such, when moving a section of the call stack to a new tab, the root will always be 100% and this value will be relative to the new total. See *Working with Tabs* for more information.
- **Function name:** Represents the short name of the function called.
- **Total time spent:** Represents the total amount of time spent in this function call. This is independent of whether the function is displayed as part of the full call stack or as a new subsection in a new tab.
- **Total number of calls:** Represents the total number of calls made to this function. This is independent of whether the function is displayed as part of the full call stack or as a new subsection in a new tab.
- **Full function name:** Represents the full function name, including assembly name. This can be configured via the Options dialog box. See *Options* for more information.

2.10 Comparing Snapshots

We often make changes in our applications and need to see how these impact performance. This can be accomplished with dotTrace Performance Snapshot Comparison feature, which displays information about the increase and decrease of timing and calls in an easy-to-follow manner, as displayed below:

```

+32.83 % Thread #1 • +698.006 ms
+32.87 % Main • +698.766 ms • -1 call • RayTraceDemo.Program.Main
+32.93 % RunMessageLoop • +700.053 ms • -1 call • System.Windows.Forms.Application+ThreadContext.RunMessageLoop(Int32, ApplicationContext)
0.00 % SetCompatibleTextRenderingDefault • -19 ms • -1 call • System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(Boolean)
0.00 % Run • -1 ms • -1 call • System.Windows.Forms.Application.Run(Form)
0.00 % EnableVisualStyles • -12 ms • -1 call • System.Windows.Forms.Application.EnableVisualStyles
0.00 % Application..ctor • 0 ms • -1 call • System.Windows.Forms.Application..ctor
0.00 % VersioningHelper..ctor • 0 ms • -1 call • System.Runtime.Versioning.VersioningHelper..ctor
0.00 % BadImageFormatException..ctor • -1 ms • -2 calls • System.BadImageFormatException..ctor
-0.06 % MainForm..ctor • -1.249 ms • -1 call • RayTraceDemo.MainForm..ctor
0.00 % WasStrongNameEvidenceUsed • -1 ms • -1 call • System.Security.Policy.Evidence.WasStrongNameEvidenceUsed
0.00 % FindType • -1 ms • -1 call • System.Security.Policy.Evidence.FindType(Type)
0.00 % get_Item • 0 ms • -3 calls • System.Collections.ArrayList+SyncArrayList.get_Item(Int32)
0.00 % get_Count • 0 ms • -5 calls • System.Collections.ArrayList+SyncArrayList.get_Count
-0.03 % ResolvePolicy • -671 ms • -2 calls • System.Security.SecurityManager.ResolvePolicy(Evidence, PermissionSet, PermissionSet, PermissionSet, PermissionSet&, Int32&, Boolean)
0.00 % SecurityManager..ctor • -4 ms • -1 call • System.Security.SecurityManager..ctor
0.00 % ProvideAssemblyEvidence • 0 ms • -1 call • System.Security.HostSecurityManager.ProvideAssemblyEvidence(Assembly, Evidence)
0.00 % RemotingServices..ctor • -4 ms • -1 call • System.Runtime.Remoting.RemotingServices..ctor
0.00 % IsAssemblyUnderAppBase • -10 ms • -1 call • System.Reflection.Assembly.IsAssemblyUnderAppBase
0.00 % CreateSecurityIdentity • -47 ms • -2 calls • System.Reflection.Assembly.CreateSecurityIdentity(Assembly, String, Int32, Byte[0..], String, Int32, Int32, Int32, Int32, Byte[0..], Evidence)
0.00 % set_DisallowBindingRedirects • 0 ms • -1 call • System.AppDomainSetup.set_DisallowBindingRedirects(Boolean)
0.00 % TurnOnBindingRedirects • 0 ms • -1 call • System.AppDomain.TurnOnBindingRedirects
0.00 % SetupDomain • -18 ms • -1 call • System.AppDomain.SetupDomain(Boolean, String, String)
0.00 % SetDefaultDomainManager • -5 ms • -1 call • System.AppDomain.SetDefaultDomainManager(String, String[0..], String[0..])
+37.14 % Thread #3 • +789.651 ms
0.01 % Thread #2 • +168 ms

```

The image displays the comparison between two snapshots. Each row displays information about the difference in call time and number of calls. If the call time and/or number of calls have increased from the first to the second snapshot, this is represented in red followed by a plus (+) sign next to the call time / number of calls. If there has been a decrease, this is represented in green and a negative (-) sign.

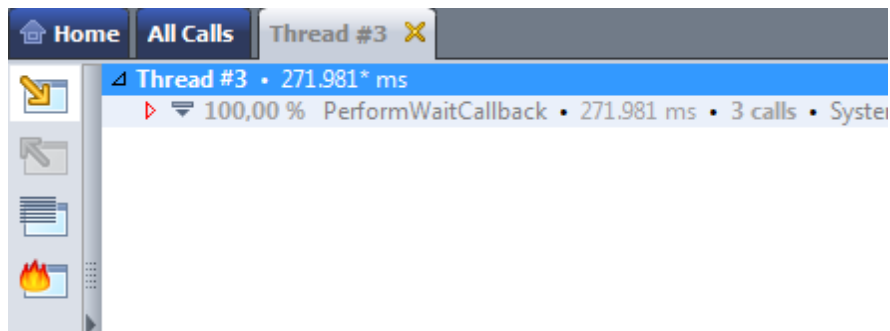
Snapshot comparisons can be done performed in multiple ways:

- Comparing tabs
- Comparing previously saved Snapshot files
- Comparing active snapshots

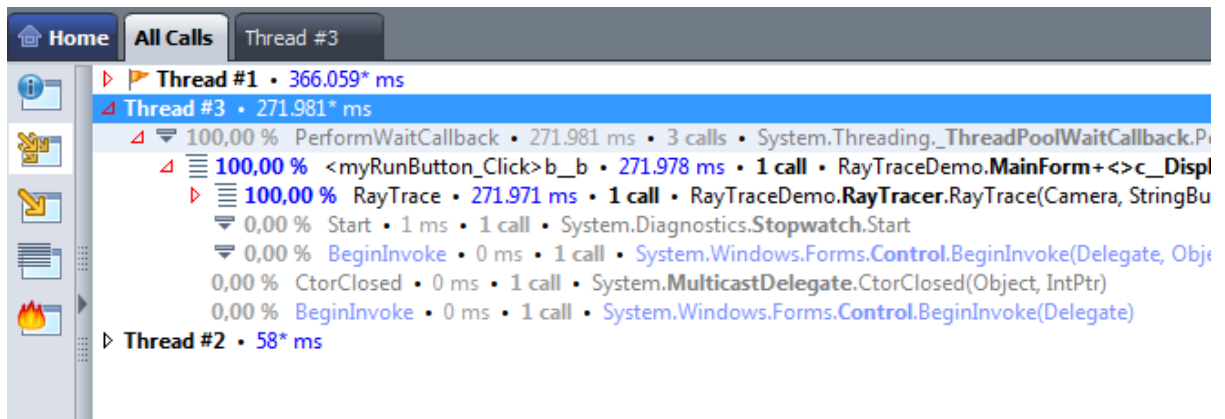
Comparing Tabs

In order to compare two tabs:

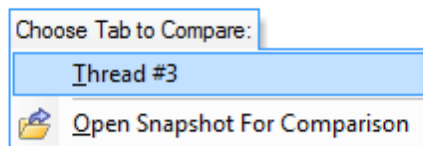
1. Select the Call or Thread in question and select **Open in new Tab** or press **Ctrl+T** (see Working with Tabs for more information)



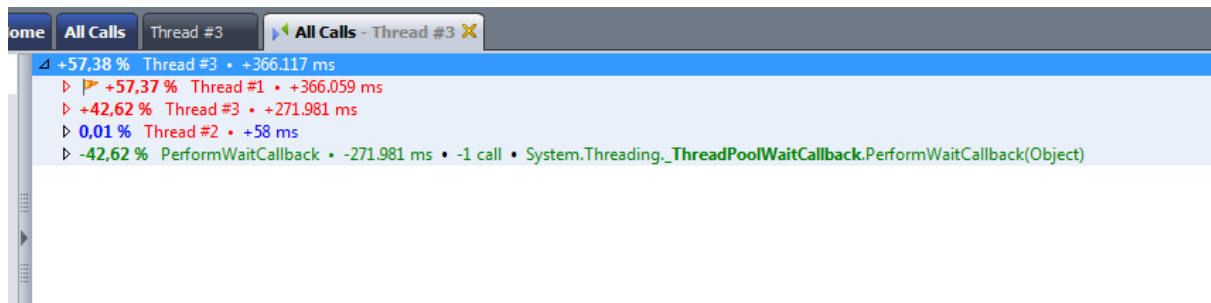
2. Click on the All Calls Tab and the thread in question (in this case it would be Thread #3). Select **File | Compare Snapshots...**



3. From the Pop-up displayed, select Thread #3



dotTrace Performance will open up the snapshot in a new Tab



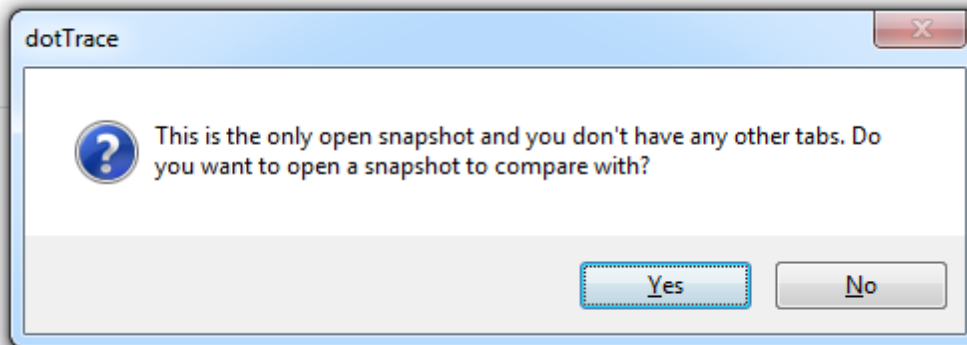
Note: The steps above only outline the process of tab comparisons. Making comparisons of tabs only has meaning when we are comparing two exact calls with manual time adjustments.

Comparing previously saved Snapshot files

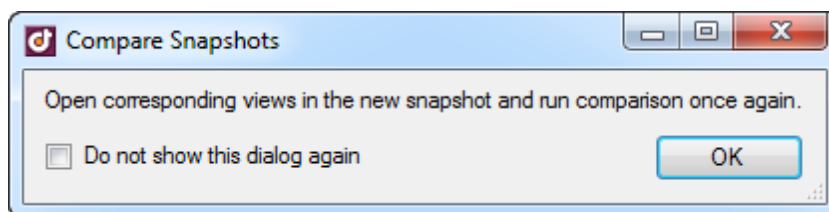
To compare two previously saved snapshots

1. Open up a snapshot using **File | Open Snapshot**

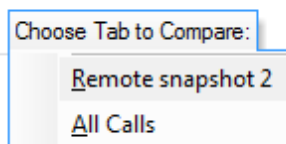
2. Select **File | Compare Snapshots....** If this is the only snapshot open, dotTrace Performance will prompt us to open a second one



3. Click on **Yes** and choose a second snapshot
4. dotTrace will open up a new instance with the snapshot and display the following message box



5. We now need to select the original snapshot and the one we are comparing to. Navigate to the first opened snapshot and select **File | Compare Snapshots...** Select All Calls from Remote Snapshot 2 (name of second snapshot).



It is important to select the correct snapshot first since based on that, the results can either be in positive or negative. For instance if Snapshot 1 is slower than Snapshot 2, then selecting Snapshot 1 first will display the results of the comparison as negative in terms of time.

Note: If multiple snapshots are already open, Steps 3 and 4 are not required.

Comparing active snapshots

If we are actively working on performance analysis and have not saved previous snapshots, we can perform snapshot comparisons easily. All that is required is to capture multiple (two or more) snapshots using the Controller panel (**Get Snapshot**). dotTrace Performance opens up an instance of dotTrace for each snapshot. We then select the first one and choose **File | Select Snapshot....** From that point on

the process is the same as comparing two previously saved snapshots.

