

ANTS Performance Profiler 5.2

December 2009

Note: these pages apply to a version of this product that is not the current released version.

For the latest support documentation, please see <http://documentation.red-gate.com>

Contents

Getting started	3
Worked example: profiling the performance of an application	4
Setting up and running a profiling session	10
Working with application settings	12
Setting up Charting Options	15
Working with profiling results	17
Working with the timeline	18
Working with the call tree	21
Working with the methods grid	25
Working with the call graph.....	27
Working with source code	31
ANTS Performance Profiler options	33

Getting started

ANTS Performance Profiler enables you to profile the code of applications written in any of the languages available for the .NET Framework, including Visual Basic .NET, C#, and Managed C++. This is useful, for example, to identify inefficient areas of your application by recording the time spent in each line of your code or method as you run your application.

You can use ANTS Performance Profiler to profile .NET desktop applications, ASP.NET web applications hosted in Internet Information Services (IIS) or the ASP.NET Development Server, .NET Windows services, COM+ server applications, Silverlight 4 or later applications, and XBAPs. In addition, you can profile applications that host the .NET Runtime, for example Visual Studio .NET plug-ins.

You can use ANTS Performance Profiler with the following versions of the .NET Framework:

- 1.1 (32-bit applications only)
- 2.0 (32-bit or 64-bit applications)
- 3.0 (32-bit or 64-bit applications)
- 3.5 (32-bit or 64-bit applications)
- 4.0 (32-bit or 64-bit applications)

ANTS Performance Profiler: step-by-step

1. Set up a new profiling session, and start profiling.
2. Optionally, select a region on the timeline to restrict the profiling results to a specific period.
3. Review the profiling results.

Worked example

Learn more about performance and memory profiling with ANTS Performance Profiler by following one this detailed example:

- [Worked example: profiling the performance of an application.](#)

Worked example: profiling the performance of an application

This worked example gives you a guided tour of ANTS Performance Profiler. It introduces you to the application's main features and shows you how you can use ANTS Performance Profiler to profile your own applications. The example shows how you can use ANTS Performance Profiler to identify the most time-critical parts of a demonstration application and determine why one programming approach is more efficient than the other.

This example is split into the following sections:

1. The Mandelbrot set demonstration application
2. Setting up the profiler
3. Profiling the application
4. Viewing the results

1. The Mandelbrot set demonstration application

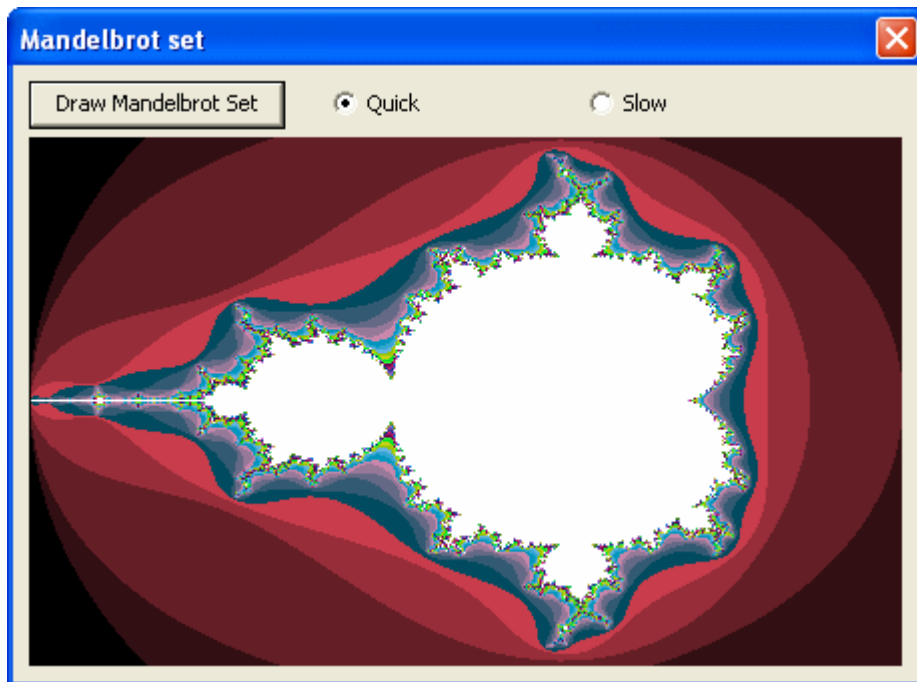
This worked example is based on a demonstration .NET desktop application that draws a fractal called the **Mandelbrot set**. A debug build of the demonstration application is included with the ANTS Performance Profiler installation. The source code for the demonstration application is also provided, along with the *.pdb* file. This is located in the same folder as the executable file, so the profiler can display source code with the profiling results.

Before you profile the demonstration application, you may wish to run the application to familiarize yourself with its behavior. If you would rather just start profiling, continue to Setting up the profiler.

1. Run the Mandelbrot set application by double-clicking the executable file located by default in the following folders:
 - ◆ For C#:
C:\Program Files\Red Gate\ANTS Performance Profiler 5\Tutorials\CS\Mandelbrot\Mandelbrot.exe
 - ◆ For Visual Basic:
C:\Program Files\Red Gate\ANTS Performance Profiler 5\Tutorials\VB\Mandelbrot\MandelbrotVB.exe

Note that this worked example refers to the C# version of the application; if you are using Visual Basic, you can follow the example, but you may see slightly different results.

2. Click **Draw Mandelbrot Set** to plot the Mandelbrot set:



The application uses two alternative methods of calculating the image. You can choose between these by selecting **Quick** or **Slow**.

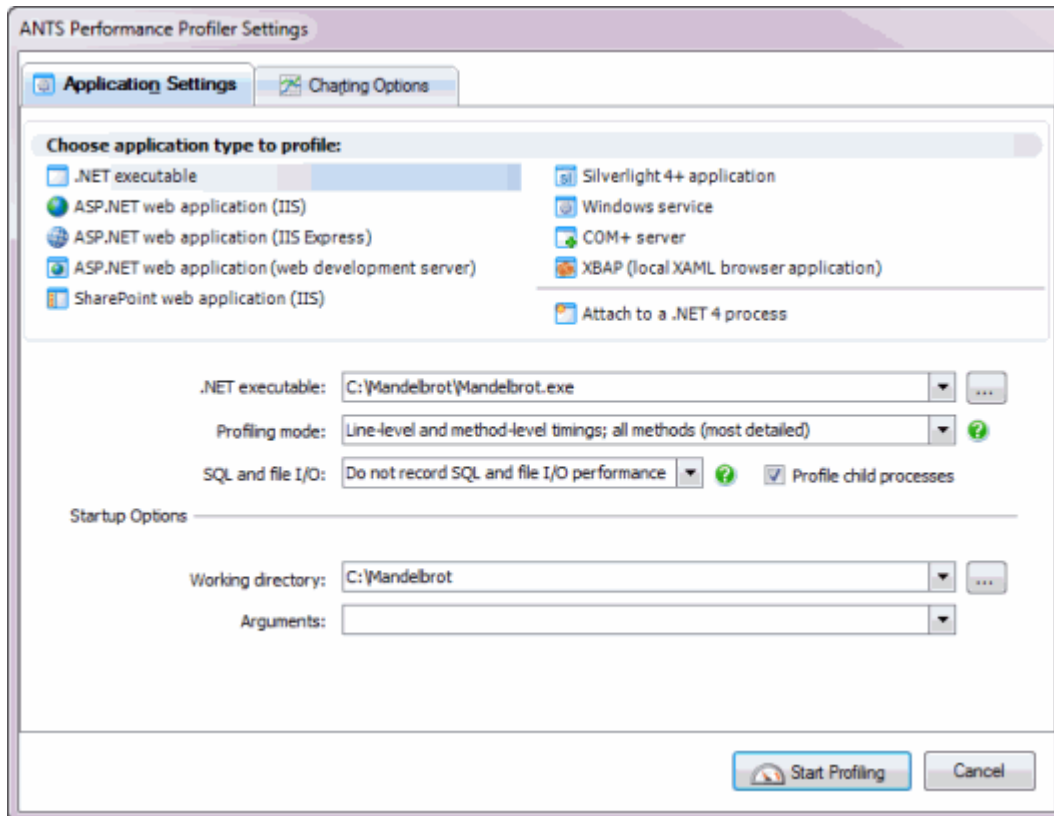
3. Close the demonstration application when you have finished using it.

To prepare for profiling the demonstration application, you first need to set up ANTS Performance Profiler.

2. Setting up the profiler


1. If you have not yet started ANTS Performance Profiler, select it from your **Start** menu.
If it is already running, on the **File** menu, click **New Profiling Session**.

The **Application Settings** tab enables you to specify the application to be profiled, set application-specific options, and choose the profiling mode.



2. Select *.NET executable* from the **Choose application type to profile** list.
3. From the **Profiling mode** list, select *Line-level and method-level timings; only methods with source (detailed)*. This profiling mode enables you to investigate how long each line of code takes to execute.

The other profiling modes are available in the professional edition of ANTS Performance Profiler. For descriptions of these other modes, see Application settings

4. Next to the **.NET executable** list, click  to browse and select the *Mandelbrot.exe*. This is located in the folder where you installed ANTS Performance Profiler, for example:

C:\Program Files\Red Gate\ANTS Performance Profiler 5\Tutorials\CS\Mandelbrot

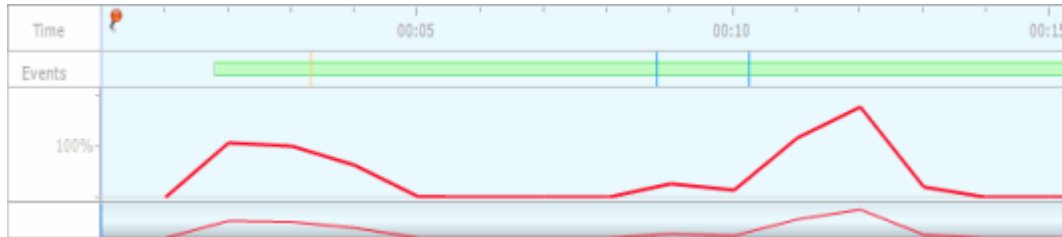
Under **Startup Options**, **Working Directory** is the directory that the application is launched from and is set to the location of the executable file. **Arguments** enables you to specify command line arguments for the running of applications. For this example, leave these settings unchanged.

3. Profiling the application

1. When you have finished setting up the profiler, click **Start Profiling** to run the Mandelbrot application and start collecting performance data.

2. In the **Mandelbrot set** window, select **Slow** to choose the slower algorithm, then click **Draw Mandelbrot Set** to draw the image.

After a few seconds, the ANTS Performance Profiler timeline is updated with performance counters and event data.

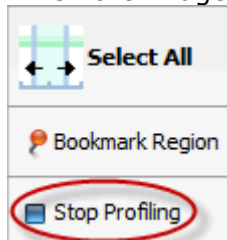


Note that execution of the Mandelbrot application is slower than usual due to the overhead of recording the profiling data. By default, ANTS Performance Profiler estimates the amount of overhead and subtracts this so that the results represent the normal execution times. This estimate is most accurate when you use a profiling mode that does not collect line-level timings. You can control whether overhead is removed from the results with the **Adjust timings to compensate for overhead added by the profiler** option (see Performance profiling options for further information).

3. Wait for a few seconds, until the *% Processor Time* performance-counter value (the red line on the timeline) reduces to zero.
4. Now, in the **Mandelbrot set** window, select **Quick** to choose the fast algorithm, then click **Draw Mandelbrot Set** to draw the image. The timeline continues to update with performance-counters and event data.



5. When the image has been drawn, click **Stop Profiling**



6. This closes the Mandelbrot application, and stops the profiling process.

4. Viewing the results

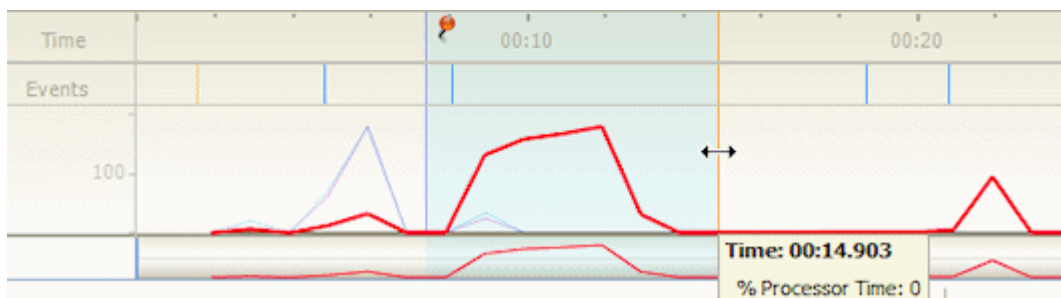
ANTS Performance Profiler summarizes the profiling results and displays the performance data as a call tree.

Method	Time (s)	Time With Children (s)	Hit Count
Mandelbrot.Form1.Main() *	<0.001	5.300	1
(Collapsed methods without source, such as framework class library methods)	0.012	3.924	4,421
Mandelbrot.Form1.CmdDraw_Click(object sender, EventArgs e)	<0.001	2.967	2
Mandelbrot.Form1.DrawMandelbrot()	0.050	2.959	2
Mandelbrot.Algorithm.Evaluate(double x, double y)	0.023	2.216	230,208
Mandelbrot.Algorithm.EvaluateUsingComplexNumbers(double x, double y) *	0.241	2.076	115,104
Mandelbrot.Complex.op_Multiply(Complex c1, Complex c2) *	0.533	1.092	1,162,876
Mandelbrot.Complex.get_X()	0.239	0.239	4,651,504
Mandelbrot.Complex.get_Y()	0.236	0.236	4,651,504
Mandelbrot.Complex.ctor(double x, double y)	0.085	0.085	1,162,876
Mandelbrot.Complex.op_Addition(Complex c1, Complex c2)	0.227	0.533	1,162,876
Mandelbrot.Complex.get_X()	0.110	0.110	2,325,752
Mandelbrot.Complex.get_Y()	0.106	0.106	2,325,752
Mandelbrot.Complex.ctor(double x, double y)	0.090	0.090	1,162,876
Mandelbrot.Complex.get_ModSquared()	0.168	0.168	1,251,796
Mandelbrot.Algorithm.EvaluateUsingDoubles(double x, double y)	0.115	0.115	115,104
Mandelbrot.Image.SetPixelColor(int i, int j, int iterations)	0.038	0.558	230,208
(Collapsed methods without source, such as framework class library methods)	0.060	0.118	1,620


The call tree displays performance data for the entire time the Mandelbrot application was running, so the results include performance data for both the **Slow** and **Quick** algorithms. For the purposes of this example, you will analyze the performance data for each algorithm individually, by selecting the appropriate region on the timeline.

You can see more information about events by moving your mouse pointer over the blue lines on the timeline events bar. A tooltip is displayed for each event, listing the event type (for example, **Click**), and other useful information such as control names and text. By reviewing this event information, you can identify relevant regions on the timeline.

1. Click and drag a region that covers the first run of the Mandelbrot application (which used the **Slow** algorithm).



ANTS Performance Profiler updates the profiling results such that they relate to the selected time period only, and displays the performance data as a call tree.

2. Click the  icon above the events bar to bookmark this region. This option is only available in the professional edition of ANTS Performance Profiler.
3. Now, repeat steps 1 and 2, but select a region that covers the second run of the Mandelbrot application (which used the **Quick** algorithm).
4. Click on the first bookmark you created to redisplay the call tree of summarized results for the first run of the Mandelbrot application (which used the **Slow** algorithm).


The call tree shows you the hottest stack trace for the selected time period (that is, the stack trace that accounts for the most execution time). It also indicates methods that may be good candidates for optimization with an asterisk (*).

5. To change the timings from percentages to seconds, on the **View** menu, click **Seconds**. Timings are displayed as percentages by default (relative to the length of the time region selected on the timeline).


The **Time With Children** column clearly shows a substantial drop in the execution time after the *EvaluateUsingComplexNumbers* method, suggesting that it may well be worth investigating this method for possible optimizations.

See Working with the call tree for more information on how to use the call tree effectively.

6. Click the *EvaluateUsingComplexNumbers* method in the call tree, to display the relevant source code. If the source code is not displayed, on the **View** menu, click **Show Source View**. You can now browse line-level timings in the source-code pane. The red bars in the heat map next to the scroll bar indicate the slowest lines of code.

7. To see a flat list of all methods hit during the selected period, click  on the display toolbar, beneath the call tree.

The methods grid is displayed. You can sort and filter methods as required. For more information about the methods grid, see Working with the Methods Grid.

8. To see a call graph of the methods hit during the selected period, select a method in either the call tree or methods grid, and click  next to the method name. The call graph is displayed.

The method you select is used as the base method for the call graph. For further information about the call graph, see Working with the call graph.

9. To display performance profiling results for the **Quick** algorithm, click on the second bookmark that you created on the timeline.

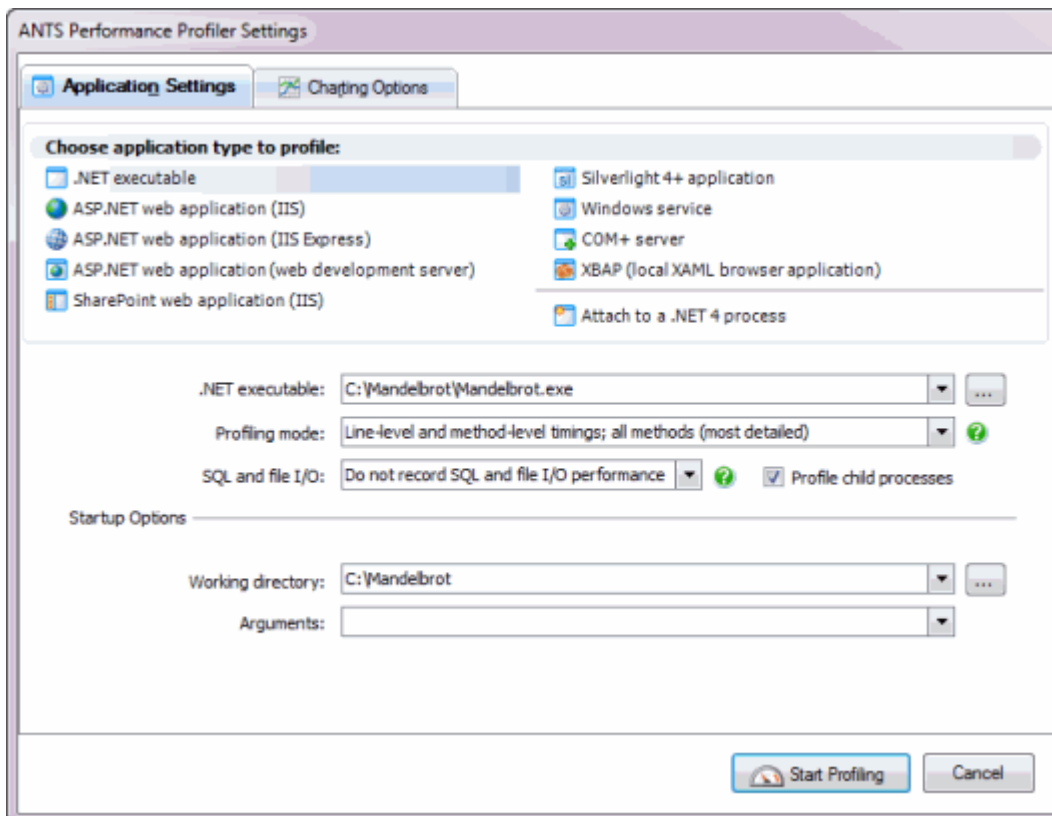
You can see from the results, that the *Evaluate* method (Time With Children) now executes much more quickly. *EvaluateUsingDoubles* replaces the inefficient *EvaluateUsingComplexNumbers* method in the **Quick** algorithm.

Setting up and running a profiling session

To profile an application, you must first set up a profiling session. A session specifies:

- The application type, location, and options for the application you want to profile.
- The profiling mode, which determines the level of detail gathered by the profiler while your application is running.
- The method used to calculate timing values (CPU time or wall-clock time).
- The performance counters to display on the timeline.

When you start ANTS Performance Profiler, the **ANTS Performance Profiler Settings** dialog box is automatically displayed; if ANTS Performance Profiler is already running, click **New Profiling Session** on the **File** menu.

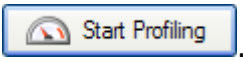


The **Application Settings** tab displays the settings for the last profiling session you ran. The settings available depend on the selected application type, and may differ from those illustrated above.

The **Charting Options** tab enables you to choose which performance counter values to display on the timeline for the new profiling session.

To set up and run a profiling session:

1. On the **ANTS Performance Profiler Settings** dialog box, complete the details on the Application Settings tab.
2. Choose which performance counters to monitor during profiling using the Charting Options tab.

3. Click .


On Windows Vista, Windows Server 2008, and Windows 7, if you are not running ANTS Performance Profiler as an Elevated administrator, the **Start Profiling** button

has a User Account Control (UAC) shield: . The UAC shield indicates that ANTS Performance Profiler will request elevation when you start profiling.

The timeline is displayed at the top of the main ANTS Performance Profiler window, and the application you want to profile is automatically started. Status text at the bottom-left of the main window indicates what ANTS Performance Profiler is doing during the profiling session.

The timeline starts displaying performance-counter data and events in near-real time. There may be a slight delay between starting a profiling session and seeing the first performance-counter data appear on the timeline.

4. To display profiling results, do one of the following:
 - ◆ Drag a region on the timeline.

Profiling data is summarized and displayed for the selected time period only. Your application will continue running and profiling will continue.
 - ◆ Click  **Stop Profiling**.

Your application will be closed. Profiling data is summarized and displayed for the entire profiling period.
 - ◆ Close your application.

Profiling data is summarized and displayed for the entire profiling period.

You can continue working with the timeline to locate periods of interest during the execution of your application, and to display the associated profiling results.

Once you have displayed some profiling data, you can view and analyze it. For more information about the different ways you can do this, see [Working with profiling results](#).

Working with application settings

Each application type is associated with a number of settings. These include settings that are common to all application types (**Profiling mode** and **Default timing display**), and some settings that are specific to individual application types.

Application types

Select the application type from the **Choose application type to profile** list. The settings available change depending on your selection:

- *.NET executable*
Startup Options. You can specify the command line arguments that are to be used when running the application.
- *ASP.NET web application (hosted in IIS)*
Server Settings and ASP.NET Account Details. If you are using IIS version 6 or IIS version 7 you can select a different port to profile on; this is not available if you are using IIS version 5. If you choose a port which is already in use, you must stop IIS to free the port before you start profiling. You can also manually specify ASP.NET account details, so that you can run a profile as a different user. This is useful if your web application needs access to a remote server. Web applications which implement the Windows Communication Foundation (WCF) can also be profiled.
- *ASP.NET web application (hosted in web development server)*
Server Settings. You can specify the URL that your web application starts on. For example, if you specify "staging" for the virtual directory and "8013" for the port number, your web application starts on URL *http://localhost:8013/staging/*.
- *Silverlight 4 browser*
Silverlight application URL. Enter the URL of the Silverlight 4 browser application you want to profile. This feature requires the Silverlight 4 plugin in Internet Explorer.
- *Windows service*
Startup Options. You can specify command line arguments that are to be used when running the application.
- *COM+ server*
Client Application. You can specify command line arguments that are to be used when running the client-application executable, and then use the client-application to test the COM+ server application during your profiling session.
- *XBAP (XAML Browser Application)*
No additional setup options are provided. To profile a remotely-hosted XBAP application, select the .NET executable application type, and then profile Internet Explorer (iexplore.exe) and navigate to the XBAP application.

- *Attach to .NET 4 process*

Choose the .NET process you want to attach to. This feature requires Windows Vista or later and .NET 4.

Profiling mode

The **Profiling mode** determines the level of detail gathered by the profiler while your application is running. The level of detail that you choose may affect the profiling speed and the overall accuracy of the results.

Profiling Mode	Speed	Accuracy	Detail	Profiling Data
<i>Line-level and method-level timings; all methods*</i>	★	★★	★★★★	All methods. This includes methods without source code, such as those in the .NET Framework class libraries.
<i>Method-level timings; all methods</i>	★★★	★★★	★★	
<i>Line-level and method-level timings; only methods with source*</i>	★★	★	★★★	Only methods for which source code is available, for example, timings will not be measured for .NET Framework methods.
<i>Method-level timings; only methods with source</i>	★★★★	★★★★	★	
<i>Sample method-level timings</i>	★★★★★	★	★	

*Profiling data is also collected for individual lines of code.

SQL and file I/O

- If you have Windows Vista or later, you can choose to record file I/O operations.
- If you have Windows Vista or later and a SQL server (except Express editions) installed on the local machine, you can choose to record file I/O operations and SQL queries.

Default timing display

The **Default timing display** determines the method used to calculate timing values:

- *CPU*

Timings *exclude* any periods of time for which a process is blocked. This can include sleeping, waiting for I/O, or waiting for some other resource.

- *Wall clock*

Timings *include* any periods of time for which a process is blocked (including sleeping, waiting for I/O, or waiting for some other resource).

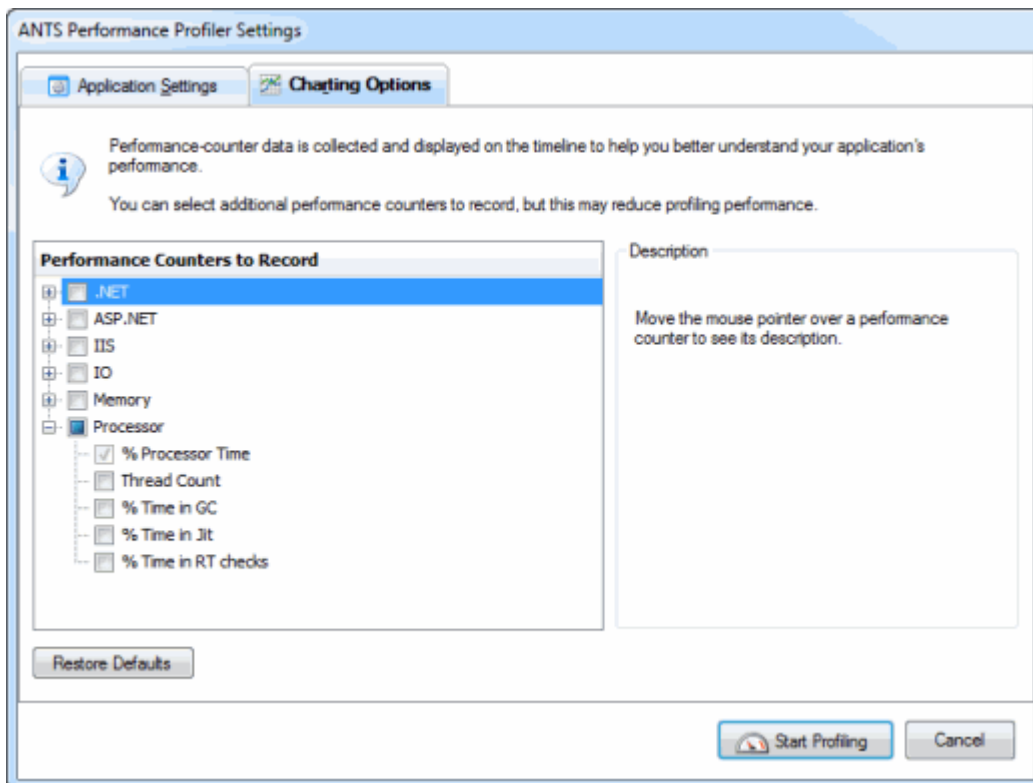
You can also change the timing display *after* you have collected profiling data, by selecting *CPU time* or *Wall-clock time* from the **Timing** list on the display toolbar. See *Working with the call tree*, *Working with the methods grid*, or *Working with the call graph* for more information.

Setting up Charting Options

ANTS Performance Profiler can monitor the values of a number of built-in Windows performance counters while the application you are profiling is executing. The values of these counters are constantly updated on the timeline as profiling proceeds.

You choose the performance counters you want to monitor using the **Charting Options** tab on the **ANTS Performance Profiler Settings** dialog box.

Note that this feature requires ANTS Performance Profiler Professional.



Not all performance counters are appropriate to all application types that you may be profiling. You can find more information about individual performance counters under the **Description** group box, including details about a counter's relevance to particular application types.

Your choice will depend on your own requirements but, as an example, you might choose the following:

From the **.NET** group:

- The *Gen 0 Promoted Bytes/sec* counter
This will give the rate at which the garbage collector promotes objects from Generation 0 to Generation 1.

From the **Memory** group:

- The *Working Set* counter
This shows the total amount of physical memory used by your service (including memory used by shared DLLs and the .NET runtime itself)

From the **Processor** group

- The *% Processor Time* counter (selected by default)
This shows the percentage of time which all running threads use on the CPU.
- The *% Time in GC* counter
This shows the percentage of time which the process was suspended to allow the last garbage collection to take place.

We recommend you avoid adding more performance counters than you need, as each additional counter that is recorded adds to the overhead introduced by the profiler. Adding too many counters may slow your application substantially.

Adding custom performance counters

You can add custom performance counters to the list of available counters in the **Charting Options** tab. To do this:

1. Close ANTS Performance Profiler.
2. Expose your performance counter to the Windows Performance Counter API using the *PerformanceCounter* and *PerformanceCounterCategory* classes of the *System.Diagnostics* namespace. (An example describing how to do this is given at <http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx> (<http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx>))

3. Create a new XML file as follows:

```
<Counters>
  <Category Name="CategoryName">
    <Counter Category="CategoryName" Name="CounterName"
Units="Measurement Units">
      <Instanced />
    </Counter>
  </Category>
</Counters>
```

Ensure that *CategoryName* and *CounterName* are the same as the names used for the *PerformanceCounterCategory* and *PerformanceCounter*. Remove the `<Instanced />` node if your counter collects data about the computer, not only the individual process. You can add multiple categories and counters in the same XML file.

4. Save the XML file as *UserCounters.xml* in *%LOCALAPPDATA%\Red Gate\ANTS Performance Profiler 6*.
5. Restart ANTS Performance Profiler.
6. The counters that you defined are shown in the list on the **Charting Options** tab.




Working with profiling results

Once you have run a profiling session and displayed some profiling results you can start analyzing the results in the results pane using the three main display types: call tree, methods grid, and call graph.

The screenshot displays the ANTS Performance Profiler interface. The top section is the **Timeline pane**, showing a graph of processor time usage over a 00:15 period. Below it is the **Results toolbar**, which includes options for 'Tree View Display Options', 'Top-down [any method]', '(All Threads)', 'CPU time', and 'Hide insignificant method'. The **Results pane (showing call tree)** displays a hierarchical view of the execution stack, starting with 'MandelbrotVB.MandelbrotForm1.Main()'. The **Find toolbar** is located below the results pane. The **Source-code pane** at the bottom shows the source code for the 'Form1.Main' method, with line 83 highlighted. The source code includes the following lines:

```
77 AddHandler Me.Closing, AddressOf Me.Form1_Closing
78 Me.ResumeLayout(False)
79 End Sub
80
81 <STAThread()> _
82 Shared Sub Main()
83 Application.Run(New Form1)
84 End Sub
85
86 Private Sub DrawMandelbrot()
87 Dim dx As Double = m_Settings.Width / m_MandelbrotI
88 Dim dy As Double = m_Settings.Height / m_MandelbrotI
```

Use the buttons on the timeline pane to switch between display types:

-  Call tree: shows stack traces that were executed by your application during the time period you have selected.
-  Methods grid: lists each method that was executed by your application during the time period you have selected.
-  Call graph: shows the calling relationships between methods executed by your application, for the time period you have selected. (The call-graph button is disabled until you have created a new call graph.)

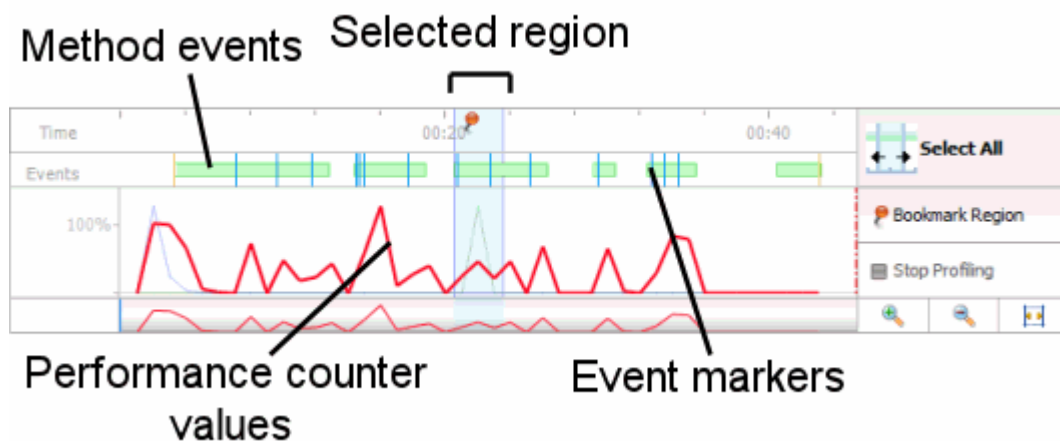
Working with the timeline

The timeline is visible throughout a profiling session, and provides a frequently updated display of performance-counter values and instances of events related to the application you are profiling. You can use this overview of application activity to isolate performance-profiling results for specific time periods.

The timeline enables you to select a region (corresponding to a time period during execution of your application) for which you wish to display profiling results. You can select and reselect any region as often as you need to, both during profiling and after you have stopped profiling and closed your application. You can also create bookmarks for selected regions, enabling you to define multiple regions and switch between them to look at data for different periods during a profiling session.

The main section of the timeline shows the values for a selection of Windows performance counters. You can choose which performance counters to display before you start profiling your application. See [Setting up performance counters](#) for more information.


The event bar on the timeline shows event markers. These indicate when certain types of event occur within your application, for example, button clicks, window activations, and exceptions. When you move the mouse pointer over an event marker a tooltip provides more information about the event. See [Working with event markers and method events](#) for more information.



Working with regions on the timeline

You can select, clear, and bookmark regions on the timeline. Whenever you select a region, profiling results are displayed that relate to the selected period only.

Selecting a region


To select a region, click and drag the mouse pointer  across the timeline. The results pane (beneath the timeline) updates to show profiling results for the selected region.


Resetting a region

To reset the currently selected region to cover the whole timeline, click **Select All**. The results pane updates to show profiling results for the entire time period for which your application was running.

Bookmarking a region




You can create a bookmark on the timeline, for a selected region. This is useful if there are several periods for which you want to view or compare profiling results: you can easily switch between bookmarked regions to redisplay profiling results.


To bookmark a selected region, click  within the selected region. This region is now bookmarked (this is indicated by a highlighted bar at the top of the timeline). To select this region again later, click within the highlighted bar.

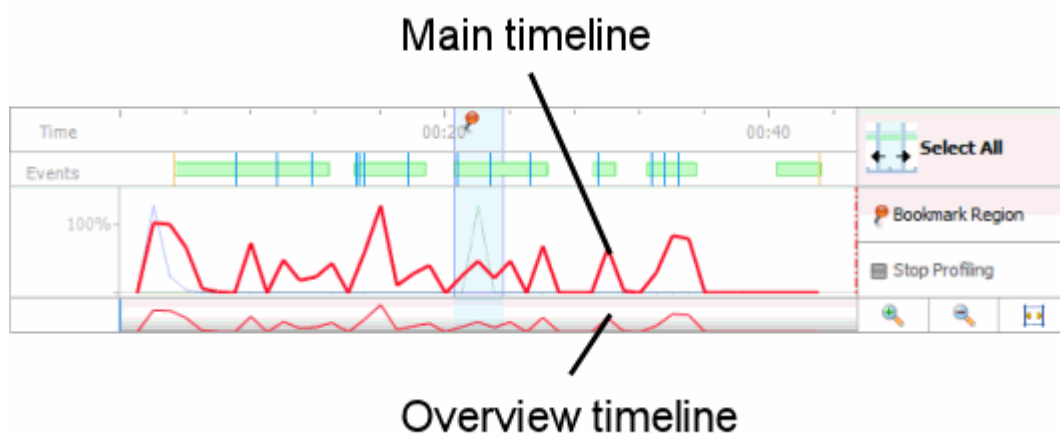
You can name a bookmark to make it easier to identify. To name a bookmark, click the highlighted bar for the bookmark and click . The name you type will be shown on the bookmark's tooltip.

To delete a bookmark, click the highlighted bar for the bookmark and click .

Adjusting the time scale

You can change the time scale to view performance-counter data in more or less detail by rotating the mouse wheel, or by using the zoom-control buttons (zoom in , zoom out , and zoom to fit ). You can also use the following keyboard shortcuts: CTRL+PLUS to zoom in; CTRL+MINUS to zoom out.

To pan the main timeline, move the mouse pointer over the highlighted area in the overview timeline () , and drag to the left or right.



Working with performance counters

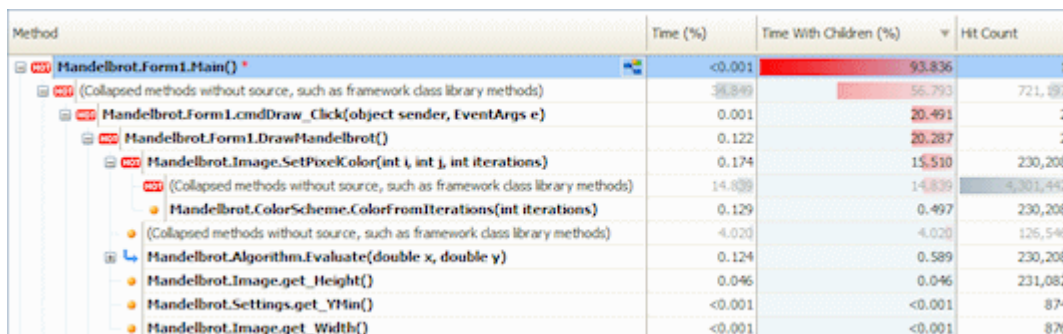
The performance counters available for the current profiling session are listed to the left of the timeline. You can choose which performance counters to display when you set up a new profiling session (see [Setting up performance counters](#)).

To highlight a particular performance counter on the timeline, click its description in the **Performance Counters** list. Values for the selected performance counter are shown on a tooltip when you move your mouse pointer over the main timeline.

Working with the call tree

The call tree shows the stack traces that were executed by your application during the time period you have selected. By default, stack traces are displayed top-down (calling method above called method). The "hottest" stack trace (the one that took the most time to run) is displayed at the top of the call tree, and is automatically expanded. If a method was called in several contexts, it is displayed once for each context in the call tree.

See Tips on using the call tree for more information on how to use the call tree effectively.



Method	Time (%)	Time With Children (%)	Hit Count
Mandelbrot.Form1.Main()	<0.001	93.836	1
(Collapsed methods without source, such as framework class library methods)	56.899	56.793	721,997
Mandelbrot.Form1.cmdDraw_Click(object sender, EventArgs e)	0.001	20.491	2
Mandelbrot.Form1.DrawMandelbrot()	0.122	20.287	2
Mandelbrot.Image.SetPixelColor(int i, int j, int iterations)	0.174	15.510	230,208
(Collapsed methods without source, such as framework class library methods)	14.839	14.839	4,301,440
Mandelbrot.ColorScheme.ColorFromIterations(int iterations)	0.129	0.497	230,208
(Collapsed methods without source, such as framework class library methods)	4.020	4.020	126,546
Mandelbrot.Algorithm.Evaluate(double x, double y)	0.124	0.589	230,208
Mandelbrot.Image.get_Height()	0.046	0.046	231,082
Mandelbrot.Settings.get_YMin()	<0.001	<0.001	874
Mandelbrot.Image.get_Width()	<0.001	<0.001	876

The following data is shown for each method within the stack trace, for the selected time period:

- **Time:** the total execution time for the method within this stack trace.
- **Time With Children:** the total execution time for the method and all its children within this stack trace.
- **Hit Count:** the number of times the method was called within this stack trace.

Each method is shown with one of the following icons:

- Root method or leaf method. Root methods are not called by any other method; leaf methods do not call any other method.
- ↳ Indicates call flow when the call-tree direction is top-down (calling method above called method).
- ↑ Indicates call flow when the call-tree direction is bottom-up (called method above calling method).
- HOT** Method is part of the hottest (longest running) stack trace. Used instead of the root/leaf or call-flow icons.

Methods listed in **bold** have source code available. To display the method's source code, click any bold method. Line-level timings are also available in the source-code pane if you use one of the *Line-level ...* profiling modes.

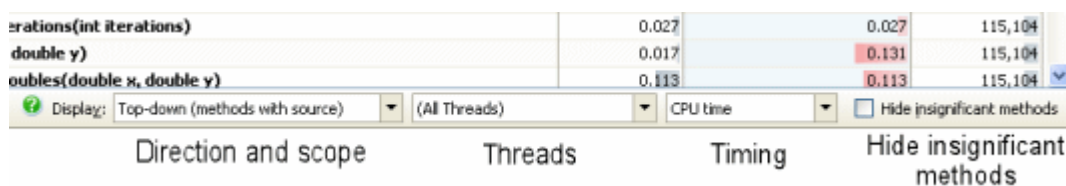
You may also see the following items in the call tree. These are shown in bold orange text, and represent time spent in your application that is in addition to time spent executing specific methods:

- **Thread blocked:** The executing thread was blocked. For example, the thread may have been sleeping, waiting for access to a shared resource, or waiting for I/O. *Thread blocked* items only contribute to timings in the call tree when the **Timing** display option is set to *Wall-clock time*. To exclude time due to *Thread blocked* items, select *CPU time* from the **Timing** display option. The call-tree display options are described below.
- **Transition to unmanaged code:** A transition from managed code to unmanaged code occurred at this point in the stack trace. In general, line-level and method-level timings are not available for the unmanaged code. However, for unmanaged methods that are declared with `extern` within managed code, method-level timings are available.
- **Transition to managed code:** A transition from unmanaged code to managed code occurred at this point in the stack trace.
- **JIT overhead:** JIT compilation occurred at this point in the stack trace during execution of your application. The method that needed to perform the compilation is shown as the parent of a *JIT overhead* item.
- **Profiler overhead:** Additional overhead introduced by ANTS Performance Profiler. This is unlikely to be seen when the option to adjust timings to compensate for overhead added by the profiler is enabled.
- **Assembly load or unload:** A .NET assembly was loaded or unloaded.
- **Module load or unload:** A .NET module was loaded or unloaded.

To create a new call graph based on a particular method, select the method in the call tree, and click the new call graph button  in the **Method** column.

Changing the call-tree display options

You can change the way data is displayed in the call tree, using the display options on the results toolbar:



- **Direction and scope:** controls whether the call tree is displayed top-down (calling methods above called methods) or bottom-up (called methods above calling methods), and also whether any method, or only methods with source, are shown. If you choose an option that shows any method, the call tree will include details for .NET Framework class-library methods.
- **Threads:** filters the display of stack traces by thread.
- **Timing:** controls the way in which method timings are calculated. You can choose from *Wall-clock time* which includes blocking such as waiting for I/O, or *CPU time* which excludes blocking.
- **Hide insignificant methods:** select this check box to hide methods that contribute less than 1% of the total execution time for the currently selected time period.

You can also:

- Change the time unit. On the **View** menu, click **Percentages, Ticks, Milliseconds,** or **Seconds.**
- Reorder the call tree. To change the stack-trace order, click the **Time With Children (%)** column heading.

Tips on using the call tree

To locate methods that may be good candidates for optimization:

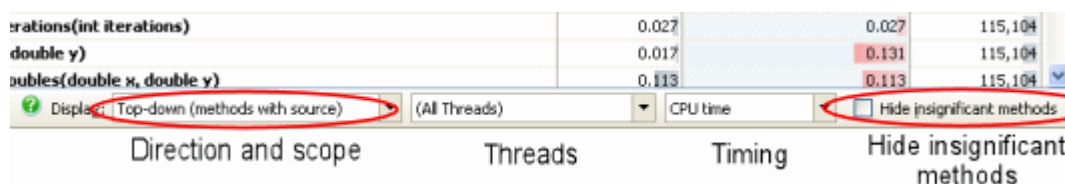
1. Order the call tree with the slowest stack traces at the top (top-down). If necessary, click the **Time With Children** column heading to change the stack-trace order.
2. Starting with the slowest stack traces, look for method pairs where subsequent values for **Time With Children** reduce substantially as you move down the stack trace. Methods with higher values in such pairs may be good candidates for optimization.

ANTS Performance Profiler can optionally suggest methods that may be good candidates for optimization. To show suggested methods, on the **Tools** menu, click **Suggest methods to optimize.** Suggested method names are marked with an asterisk (*).

In general, the better you understand the structure and meaning of your code, the more easily you will be able to interpret the data collected by the profiler.

To reduce the number of methods shown, you can do any of the following:

- Choose a "(*methods with source*)" option from the **Direction and scope** list in the display options.



- Select the **Hide insignificant methods** check box in the display options.
- Select a shorter region on the timeline.



To find a particular method:

1. On the **Tools** menu, click **Find**.

The **Find** bar is displayed beneath the call tree.

2. Type all or part of the method name you are looking for, and press ENTER.

The first matching row in the call tree is highlighted.

Click  **Previous** or  **Next** to move between matching method names.

Working with the methods grid

The methods grid lists each method that was called by your application during the time period you have selected. Even if a given method is called in several contexts, it is shown only once in the methods grid, with aggregated data that accounts for all contexts. You can order the data by any column by clicking the column heading. Data is ordered by **Time With Children** by default.

Namespace	Method Name	Time (%)	Time With Children (%) ▾	Hit Count	Source File
Mandelbrot	Form1.Main()	0.000	94.495	1	Form1.cs
Mandelbrot	Form1.ctor()	0.003	40.333	1	Form1.cs
Mandelbrot	Form1.cmdDraw_Click(object sender, EventArgs e)	0.001	21.003	2	Form1.cs
Mandelbrot	Form1.DrawMandelbrot()	0.082	20.746	2	Form1.cs
Mandelbrot	Image.SetPixelColor(int i, int j, int iterations)	0.168	16.267	230,208	Image.cs
Mandelbrot	Form1.InitializeComponent()	0.007	7.750	1	Form1.cs
Mandelbrot	Form1.Dispose(bool disposing)	0.000	1.520	1	Form1.cs
Mandelbrot	Algorithm.EvaluateUsingDoubles(double x, double y)	0.393	0.393	115,104	Algorithm.cs
Mandelbrot	ColorScheme.ColorFromIterations(int iterations)	0.107	0.308	230,208	ColorScheme.cs
Mandelbrot	Image.ctor(int width, int height)	0.001	0.116	1	Image.cs
Mandelbrot	Algorithm.Evaluate(double x, double y)	0.117	0.113	230,208	Algorithm.cs
Mandelbrot	Image.get_Height()	0.032	0.032	231,082	Image.cs

The following data is shown for each method, for the time period you have selected:

- **Time:** the total execution time for the method (in all contexts).
- **Time With Children:** the total execution time for the method and all its children.
- **Hit Count:** the number of times the method was called.

Methods listed in **bold** have source code available. To display the method's source code, click any bold method. Line-level timings are also available in the source-code pane if you used the *Line-level ... (most detail)* profiling mode.

Changing the methods-grid display options

You can change the way data is displayed in the methods grid, using the display options on the results toolbar.

ations(int iterations)	0.027	0.027	115,104
double y)	0.017	0.131	115,104
oubles(double x, double y)	0.113	0.113	115,104

Display: Top-down (methods with source) ▾ (All Threads) ▾ CPU time ▾ Hide insignificant methods

Scope Threads Timing Hide insignificant methods

- **Scope:** controls whether any method, or only methods with source, are shown. If you choose to display any method, the methods grid will include details for .NET Framework class-library methods.
- **Threads:** filters the display of stack traces by thread.
- **Timing:** controls the way in which method timings are calculated. You can choose from **Wall-clock time** which includes blocking such as waiting for I/O, or **CPU time** which excludes blocking.
- **Hide insignificant methods:** select this check box to hide methods that contribute less than 1% of the total execution time for the currently selected time period.

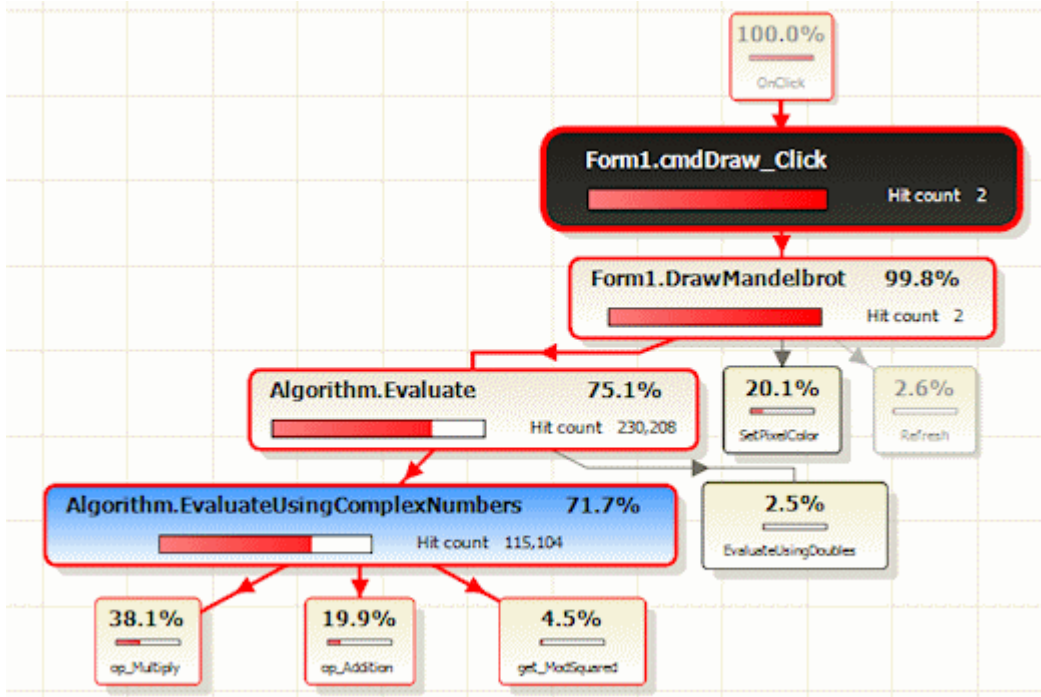
Tips on using the methods grid

To find particular methods:

1. On the **Tools** menu, click **Find**.
The **Find** bar is displayed beneath the methods grid.
2. Type all or part of the method name you are looking for.
As you type, the methods are filtered to display only those that match your text.

Working with the call graph

The call graph shows the calling relationships between methods during the execution of your application, and is focused on a method of your choice (the *base* method; shown in black in the example below). If a given method is called in several contexts, it is shown once for each context in the call graph. The base method is shown only once in the call graph, unless it is called recursively.




Selecting a base method makes it easy for you to visualize all the callers and callees for that method.

The percentage value shown in each method is calculated with respect to the base method as follows:

- For a method called by the base method, this is the percentage of the base method's execution time that the method accounts for, relative to the base method's total execution time.
- For a method that calls the base method, this is the percentage of the base method's total execution time that is due to the calling method.

Calculations are always made with respect to the selected region on the timeline, or the whole profiling period if you have not selected a region.

Creating a new call graph

Every instance of a call graph is based on a particular method, so you must first select a method in the call tree, methods grid, or source code, then click the create new call graph button :

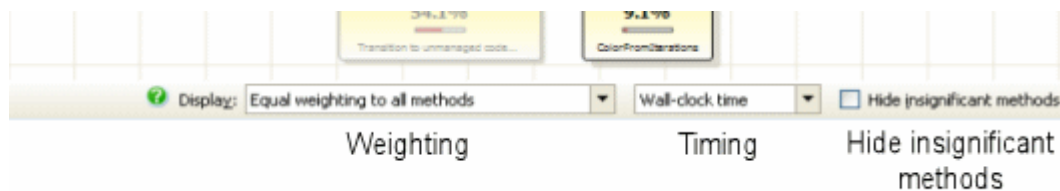
	0.007	9.358	1
ibrary methods)	1.591	45.253	1,718
EventArgs e)	0.001	13,302	1
	0.063	13,101	1
int iterations)	Create a new call graph for this method.		9.711 115,104
work class library methods)	1.280	2.593	810
	<0.001	7.405	1

Alternatively, right-click the method and select **Create new call graph** on the short-cut menu.

The call graph is displayed in the results pane.

Changing the call-graph display options

You can change the way data is displayed in the call graph, using the display options on the results toolbar:



- **Weighting:** controls the way that methods are drawn on the call graph. *Equal weighting to all methods* is useful when you need to see how methods without source code (for example, .NET Framework library methods) affect the execution times in your application. *Emphasize methods with source* draws the call graph with much smaller boxes for those methods that do not have source code available. This allows you to concentrate on the timings for those methods for which you have the source code.
- **Timing:** controls the way that method timings are calculated. You can choose from **Wall-clock time** which includes blocking such as waiting for I/O, or **CPU time** which excludes blocking.
- **Hide insignificant methods:** select this check box to hide methods that contribute less than 1% of the total execution time (for the currently selected time period).

You can also change the time unit. On the **View** menu, click **Percentages**, **Ticks**, **Milliseconds**, or **Seconds**.

Navigating the call graph

You can resize the call graph by rotating the mouse wheel, or by using the zoom controls to the left of the call graph. You can pan the call graph by clicking and dragging on a blank part of the graph.

To expand a method on the call graph (that is, to show the method's immediate children or parents), click the method.

To expand the most expensive path from a particular method, hold down the CTRL key and click the method. Alternatively, right-click and select **Expand most expensive stack trace**.

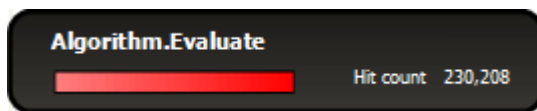
To expand the most expensive path for all children from a particular method, hold down the SHIFT key and click the method. Alternatively, right-click and select **Expand most expensive stack trace for all callees**.

To expand the most expensive path for all parents of a particular method, hold down the SHIFT key and click the method. Alternatively, right-click and select **Expand most expensive stack trace through all callers**.

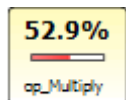
To collapse a method on the call graph, double-click the method.

More about call graphs

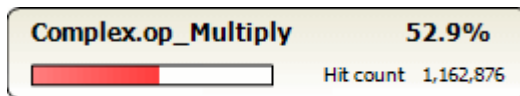
Methods are drawn in several different styles in a call graph:



Base method (the method you chose when creating the call graph). Execution-time percentages are calculated with respect to this method.

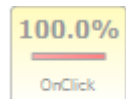


(Minimized)

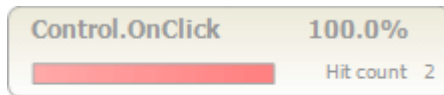


(Maximized)

Method with source code.

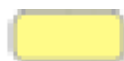


(Minimized)

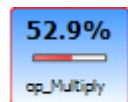


(Maximized)

Method without source code. This style is used when all methods have equal weighting.



Method without source code. This style is used when methods with source code are emphasized.




(Minimized)



(Maximized)

Selected method. When a method is selected, *all* methods in that stack trace are also outlined in red.



Recursive method. The  symbol is added to any method that is called recursively within your application.

Call graphs always include methods for which no source code is available (for example, methods from the .NET Framework class library) and methods for all threads running in your application during profiling.

It is not possible to change the time period covered by an existing call graph. To create a call graph for a different time period, return to the call-tree or methods-grid display, reselect the required period on the timeline, and create a new call graph.

Working with source code

When you select a method in the results pane and source code is available for that method, the code is displayed in the source-code pane with the first line of the method body highlighted.

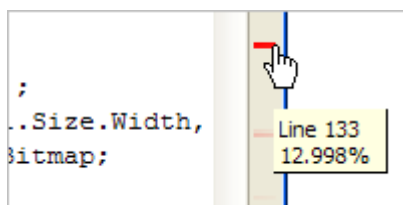
Line	Hit Count	Avg Time (%)	Time (%)	Code
29				
30				public override string ToString()
31				{
32				return(String.Format("{0} + {1}i", X, Y));
33				}
34				
35				public static Complex operator +(Complex c1, Complex c2)
36				{
37	1,162,876	0.000	12.432	return new Complex(c1.X + c2.X, c1.Y + c2.Y);
38				}
39				
40				public static Complex operator *(Complex c1, Complex c2)
41				{
42	1,162,876	0.000	21.370	return new Complex(c1.X * c2.X - c1.Y * c2.Y, c1.X * c2.Y + c1.Y * c2.X);
43				}
44				
45				/// <summary>
46				/// Real part.

If you used the most detailed profiling mode for your profiling session, *Line-level ... (most detailed)*, line-level timings are shown for each line of code (as well as average time and hit count).



Navigating through source code

You can navigate through the source code for a particular file in several ways:

- To jump directly to lines of code that accounted for the most execution time, use the heat map alongside the vertical scroll bar. Colored bars indicate the location of the slowest lines of code in the source-code file:



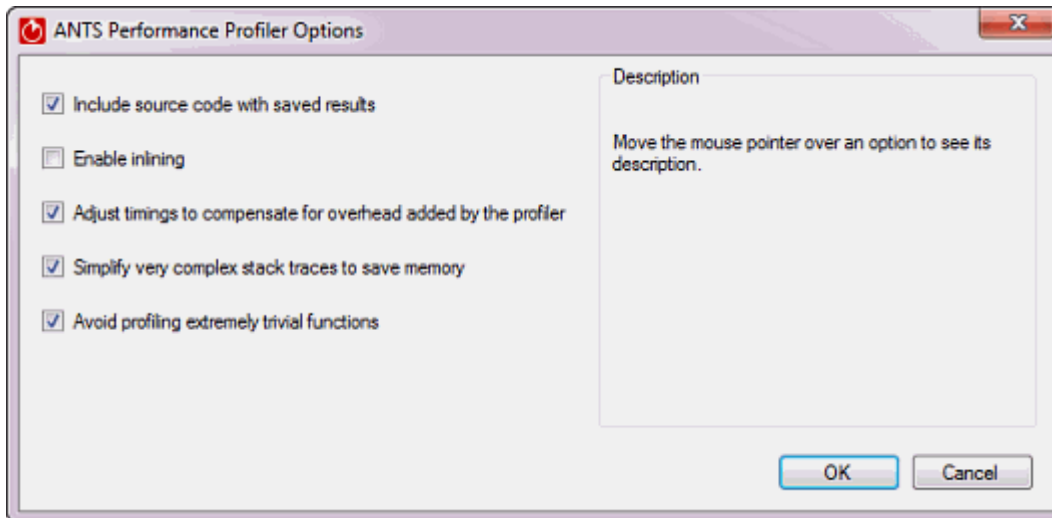
Click a bar to jump to the relevant line. The heat map is not available if you did not collect line-level timings.

- To jump directly to a particular method, select the method from the list directly above the source code. A colored bar against each method indicates the relative time spent within the method.
- To jump between methods from within the source code, click the method call. This click-through navigation works for most method calls where the called method has source code available.
- Use the forward  and back  history buttons.

You can also create a new call graph from within the source-code pane: right-click the method that you want to base the call graph on, and select **Draw Call Graph** > *<method name>*.

ANTS Performance Profiler options

ANTS Performance Profiler includes a number of options that are applied to all profiling sessions. To access these options, on the **Tools** menu, click **Options**.



Unless you have a particular need to adjust the options, leave them at their default settings. Changing the default setting for certain options may cause problems during profiling.

Include source code with saved results

Enable inlining

Adjust timings to compensate for overhead added by the profiler

Simplify very complex stack traces to save memory

Avoid profiling extremely trivial functions

Include source code with saved results

Includes the contents of source files when you save profiling results. This means that you can review line-level performance data in saved results, without having to restore your source files to their original state.

You may want to clear this option if, for example, you need to distribute performance profiling results for an application that has confidential source code.

By default, this option is selected.

Enable inlining

Enables inlining of methods by the .NET JIT compiler, for the process being profiled.

If you are profiling the release build of an application, selecting this option will produce a profile that is closer to the "real-world" performance. However, the accuracy of the results will be reduced. In particular, line-level timings will be distorted, hit counts will not be recorded for inlined methods, and time spent in inlined methods will be reported as part of the calling method.

By default, this option is not selected.

Adjust timings to compensate for overhead added by the profiler

Adjusts timings by estimating the influence the profiler has had on the process being profiled, and subtracts this from the profiling results. This estimate is most accurate when you use a profiling mode that does not collect line-level timings.

The design of modern processors means that this estimate may not always be accurate, especially for short function calls.

By default, this option is selected.

Simplify very complex stack traces to save memory

Summarizes complex stack traces in profiling results. This conserves resources on the machine you are using for profiling. The stack traces that are summarized are unlikely to be important to your profiling results. However, if you wish to see these summarized results, you can clear this option.

Clearing this option can significantly increase the memory required by the profiler. Depending on the application you are profiling, the profiler may become unstable if you clear this option.

By default, this option is selected.

Avoid profiling extremely trivial functions

Prevents profiling of methods that have a running time measured in tens of nanoseconds, and which contribute to less than one-billionth of the run time in total. Typically, these methods do not produce very relevant performance data. Ignoring these methods reduces the amount of memory required to store and process profiling results.

By default, this option is selected.