



High-Profile Profiling

Introduction

While developing your applications you always use various sophisticated tools well suited for a specific kind of job. Why not use appropriate techniques for optimizing your software? We often end up with bloated applications merely because we refuse to give a consideration to all the possibilities profiling tools offer. However, the only result such neglect can bring is a grief over perfectly optimized chunks of program where it spends less than 1% of the overall runtime, while major performance bottlenecks are left intact.

Being a professional programmer you naturally possess the ability to constantly keep in mind optimization issues, while working on the piece of code critical to your application. But imagine a typical situation, when you honestly try to analyze code and subsequently implement an optimization strategy every time your experienced eye comes across some gross coding error or just barely distinguishable “code smell”. After all these refactorings, amendments and minor alterations you might eventually come to a decision that you simply cannot have any memory management or performance issues outstanding. How frustrating it might be then, to learn that all your efforts have been applied to the portion of code that doesn’t require optimization to any extent. To put it simply: how can I be sure that the most attention has been paid to the most “insatiable” and “weak” parts of the program in terms of memory usage and performance?

If you have a tool in your arsenal that lets you be perfectly aware of every optimization issue from your code’s “hot spots” to objects’ lifespan, time spent on tracking down and tackling memory retention problems and performance bottlenecks pays off very quickly and to the great extent, especially when you plan on extending your program’s functionality in the future.

Our point is – the analysis process aimed at code optimization should be intentionally segregated from the process of actually optimizing code. *Our proposed solution is* - super fast and intuitive dotTrace profiler, which aims to help you attain better understanding of how well your applications deal with memory management and where potential performance problems may lie.

With the major benefits of profiling outlined, let’s get to our simple yet quite illustrative example.

In our example we are going to use the dotTrace profiler. The latest version of the tool is always available under <http://www.jetbrains.com/profiler/download/index.html>.

Starting up the profiler

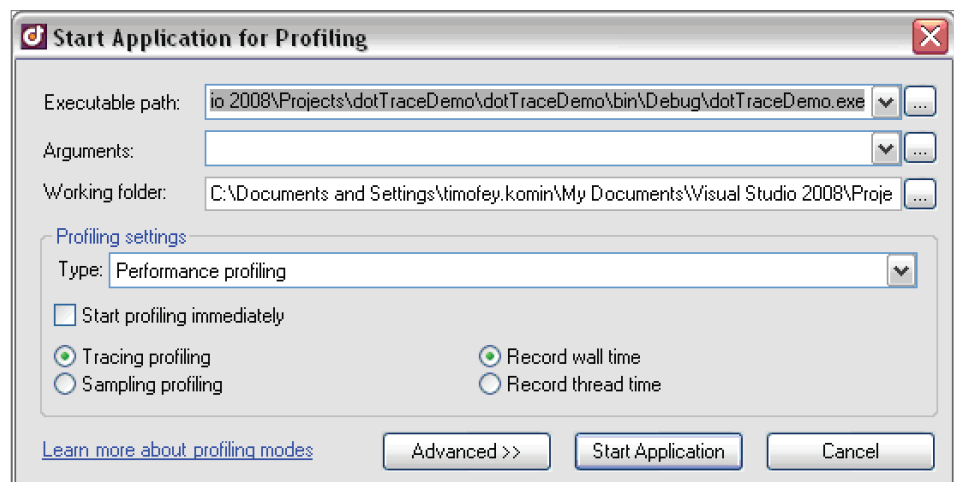
After installing the profiler you basically can choose between two work patterns:

- Using the profiler as a stand-alone application;
- Invoking the profiler from within Visual Studio.

If you launch dotTrace in stand-alone mode, you will be presented with a handy welcome screen offering you to immediately start profiling applications of your choice.



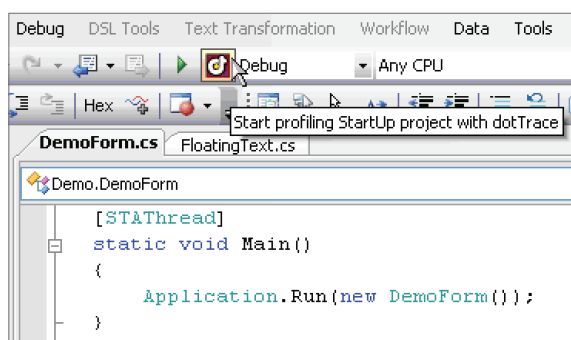
After clicking the **Profile Application** link the **Start Application for Profiling** dialog is displayed.



Specifying the executable path and working folder is essential. Custom profiling settings corresponding to your needs can be set in this dialog as well. Detailed information on all the available options can be found under the **Learn more about profiling modes** link.

As already mentioned, the second work pattern available to you is invoking the profiler right from within Visual Studio. In the process of installation dotTrace integrates seamlessly into Visual Studio environment adding the **Start profiling with dotTrace** button right next to the **Start Debugging** button on the standard toolbar.

While working on some project, you can quickly start profiling your application by clicking the **Start profiling with dotTrace** button and then switch back to Visual Studio when the profiling session ends.



Upon clicking this button you will be presented with the **Start Application for Profiling** dialog, where all required fields are already filled in.

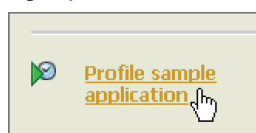
The profiling session

A typical profiling session consists of the following steps:

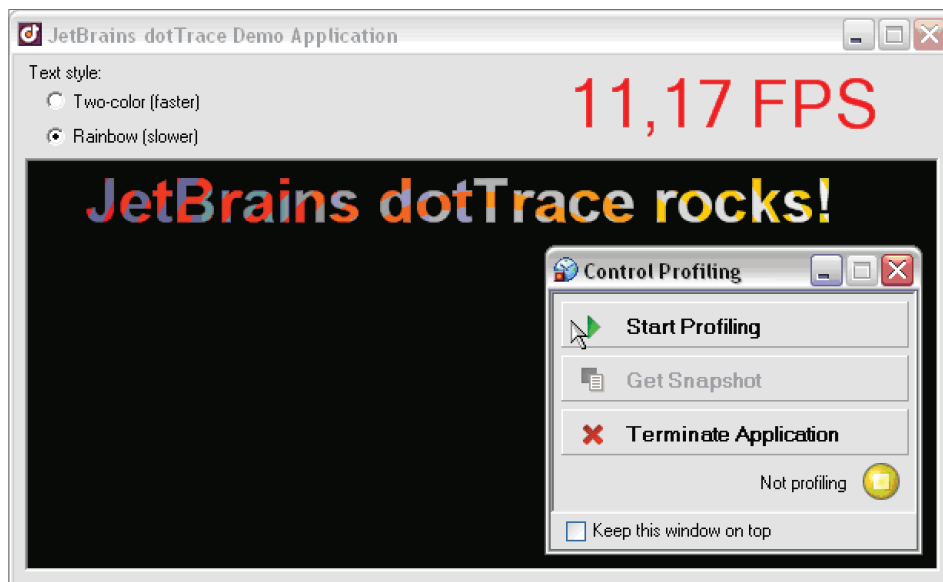
- Running the application to be profiled
- Performing necessary steps on the target application
- Taking a snapshot
- Locating a bottleneck
- Outlining an optimization strategy

We are going to illustrate all these steps by performance profiling a simple application distributed along with the profiling tool itself.

After starting up dotTrace, the **Profile sample application** link appears in the right part of the main window.



Upon clicking it, the sample application is invoked and the profiling control panel appears:

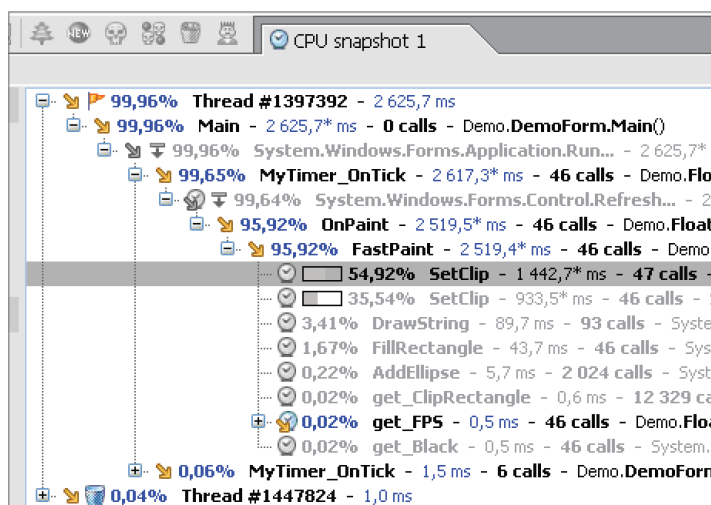


After you click the **Start Profiling** button dotTrace starts gathering profiling data. The application we are profiling in this example is rather simple, so we can just wait a couple of seconds to let dotTrace gather some statistics and then immediately click the **Get Snapshot** button.

In real-life situations you might need to perform some actions on the target application before taking a snapshot. These actions should correspond to the feature you are profiling.

When you click the **Get Snapshot** button, dotTrace's main window becomes active and the performance snapshot is loaded.

By default, the snapshot is loaded in a call tree view. The call tree represents the relationships between callers and callees for each thread of your application.

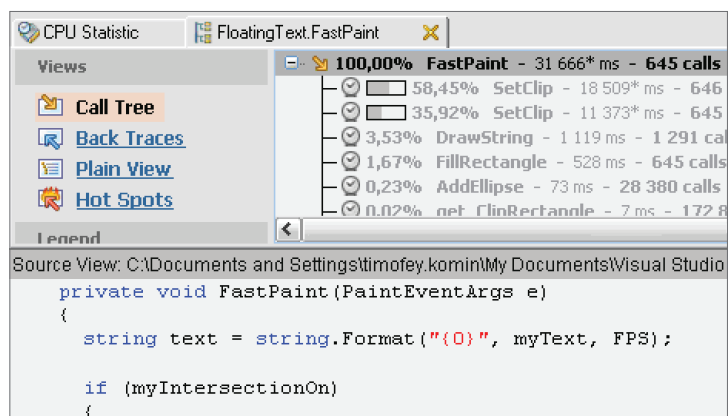


Names of the methods are displayed along with statistical information. While navigating through the call tree, you can review such parameters as the number

of calls to a particular method, absolute time spent within a method, percentage of time consumed in relation to the overall profiling time. Calls to system methods are filtered (grayed out in GUI) presuming that there is little need in investigating their performance.

The call tree view is eminently useful for locating performance bottlenecks:

Drilling into the call tree, you will almost surely notice that roughly 96 percent of time (in our case) was consumed by the **FastPaint** method. This looks like a major bottleneck of our application. Now that we have located it, we can thoroughly examine performance of the **FastPaint** method by opening it in a separate tab using a simple command (**Right Click | Open in New Tab**) or a handy shortcut (**Ctrl + T**).



Percentage	Method	Time (ms)	Calls
100,00%	FastPaint	31 666*	645 calls
58,45%	SetClip	18 509*	646
35,92%	SetClip	11 373*	645
3,53%	DrawString	1 119	1 291 ca
1,67%	FillRectangle	528	645 calls
0,23%	AddEllipse	73	28 380 calls
0.02%	net.ClipRectangle	7	172 f

```

Source View: C:\Documents and Settings\timofey.korin\My Documents\Visual Studio
private void FastPaint(PaintEventArgs e)
{
    string text = string.Format("{0}", myText, FPS);

    if (myIntersectionOn)
    {

```

Time spent within **FastPaint** method is now taken as 100%. Methods called during **FastPaint** execution are sorted by percentage value of time consumed. This helps us instantly estimate, how much of the bottleneck can be eliminated by optimizing major “time consumers” within our bottleneck.

To deliver an even more detailed picture is the source code pane located right below the call tree. It automatically scrolls to necessary methods as you wander up and down the call tree and lets you make instant assessments on problem areas of the program. If you invoke dotTrace from within Visual Studio, the **Open in Visual Studio** link (upper right corner of the source code pane) brings you back to VS with caret standing at the first code line of the corresponding method.

Conclusions

It took just a couple of minutes to spot the portion of code upon which all future efforts should be concentrated. Based on gathered profiling data, we have spotted the performance bottleneck in just a few clicks *with* pinpoint accuracy and *without* making any unnecessary and potentially erroneous assumptions.

Sometimes it may well be so, that technical expertise and experience should not serve problem solving but be used to rationally select tools that will solve the problem for you.

The dotTrace profiler is of course not only about performance profiling. It also has powerful capabilities in memory profiling, offering such useful features as memory snapshot comparison and allocations gathering.