

Win in the flat world

Optimizing Application Performance with Visual Studio 2005

Tuning the performance of .NET 2.0 applications through the project lifecycle

– Atul Gupta

To ensure optimum application performance in terms of throughput and processing speed, performance tuning can play a critical role. However, it could entail risks and unforeseen costs without the right approach.

In order to reduce risks and costs, performance tuning should be treated as an integral part of a project life cycle. In most projects, however, it is conducted during the final stages of testing or when customers complain about performance post production deployment.

Performance tuning of .NET 2.0 applications can be achieved using Visual Studio 2005 tools. In this paper, we explore these tools and also provide guidelines on how to use them.



Introduction

The activity of performance tuning identifies bottlenecks in the code and environment (application configuration files, hardware configuration) of application development. Addressing these bottlenecks ensures that the application meets the required performance criteria such as transactions/sec, requests/sec, web page load time, and so on.

Thus, the key to reducing risks and total cost of ownership in application development is persistent and diligent performance tuning.

To avoid unanticipated performance issues toward the end of the project life cycle, it is essential to commence performance tuning at the design stage itself. For example, addressing a design issue (to enhance the application's performance) at the eleventh hour – in the testing stage – will impede the project completion greatly.

A software application should ideally meet the following two primary criteria:

1. Functional Requirements
2. Non-Functional Requirements (NFR) / Quality of Service (QoS) requirements.

Organizations generally have robust processes in place to comply with functional requirements. However, they often lack a proper approach in dealing with NFRs like performance.

In many projects, NFRs (availability, scalability and so on) other than performance are given greater importance. This may not be the right approach, as managing availability and reliability requirements would be irrelevant if the application is unable to handle even a small user load (say five concurrent users). To achieve optimal efficiency of any process or operation, ensuring the best performance of an application is vital.

During performance tuning, you will also need to deal with the effect that some NFRs have on the performance of the application. NFRs such as extensibility, high security and flexibility lower the performance of an application.

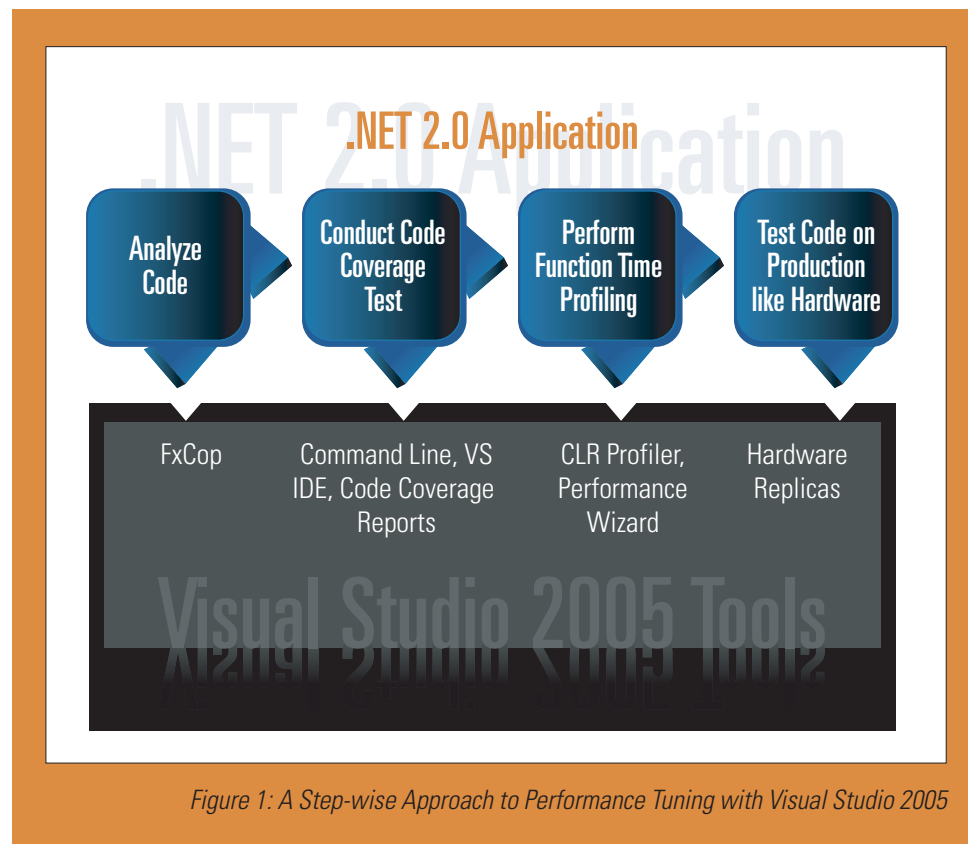
Thus, clear understanding, prioritization and threshold level identification of NFRs play substantial roles in ensuring the successful execution and delivery of an application.

This paper highlights important processes that you should follow while developing an application with Visual Studio 2005 (VS 2005). Adhering to these guidelines will ensure successful performance tuning of .NET 2.0 applications.

An Outline

Performance Tuning with Visual Studio 2005

The performance tuning process of .NET 2.0 applications using VS 2005 is broadly divided into four steps. The following diagram provides an overview of these steps:



In the following sections, a detailed description of each step is provided.



Analyze Code

This is an elementary step in the process of analyzing code for bottlenecks. Widely used during code analysis, FxCop provides recommendations for managed assemblies. Earlier a standalone application, it is now built into VS 2005. One of the rule libraries in FxCop contains performance-related checks. Some rules in the performance rule set are as follows:

1. Do not cast unnecessarily
2. Remove unused locals
3. Prefer jagged arrays over multidimensional
4. Avoid unnecessary string creation
5. Dispose methods should call **SuppressFinalize**

Note: This is not an exhaustive list. Do refer to the MSDN documentation that comes with VS 2005 for a complete listing.

To access FxCop from VS 2005, do the following:

1. Right-click on a project in the **Solution Explorer** window. Alternatively, select the **Build** menu in VS IDE.

2. From the menu, select **Run Code Analysis**.

3. FxCop executes and displays the results that are listed as warnings along with a description and location of the issue (file name, line number, and so on)

To run FxCop on every build, do the following:

1. In the **Project Properties** window (as shown in the figure below) click the **Code Analysis** tab. The **Code Analysis** window displays.

2. In the **Code Analysis** window, select **Enable Code Analysis (defines CODE_ANALYSIS constant)** checkbox.

3. From the **Rules** list, you can additionally select the rules to execute against your code.

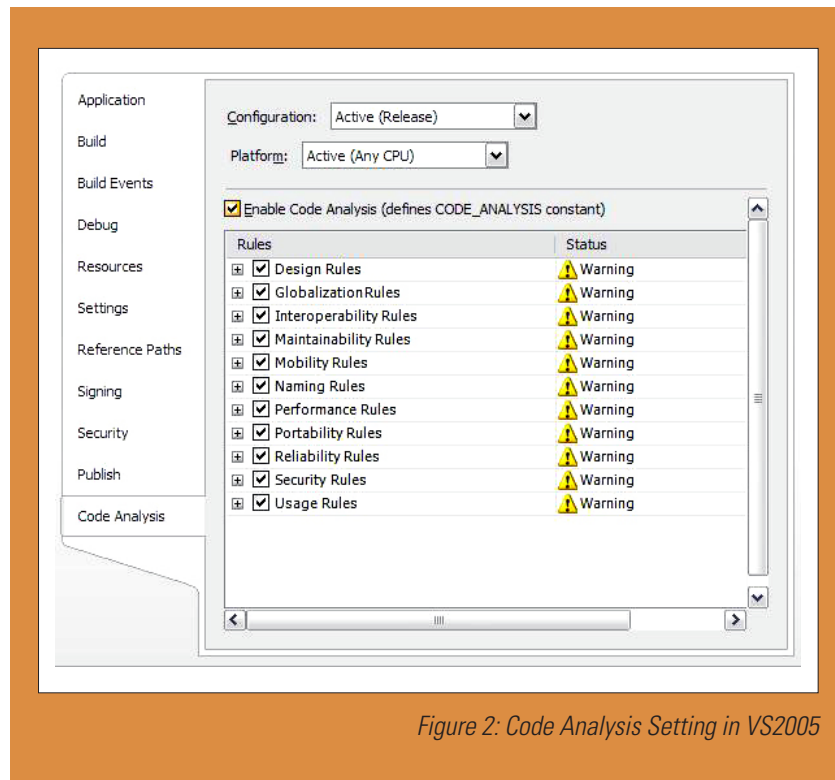


Figure 2: Code Analysis Setting in VS2005

Note: FxCop provides only an elementary list of code issues. In order to prioritize these issues, it is essential to perform function time profiling. For instance, consider a function that takes 100 seconds to execute. The FxCop report may provide several suggestions to decrease the execution time of this function. One of the suggestions could be, say, the usage of StringBuilder class instead of string concatenation. If this suggestion is implemented prior to function time profiling, it may reduce the execution time of this function by a marginal value, say, 0.1 sec (i.e. 0.1 percent). However, function time profiling could reveal that implementing another suggestion of FxCop would have reduced the execution time by, say, as much as 5 percent. Thus resolving an issue prior to performing function time profiling could well be a waste of time and effort.

Thus, correcting the issues identified by FxCop is recommended after performing [Step 3 – Perform Function Time Profiling](#)



Conduct Code Coverage Test

Code evolves through various stages of a project and the amount of code churn in the initial stages of a project is exceptionally high. Changing requirements, varied algorithms (used to solve problems and hence continuously refined), numerous developers working on the same piece of code, etc, contribute to code churn. This results in the generation of redundant code and variables.

Code coverage tests identify redundant sections of code, highlight sections of code skipped by test plans, identify potential bugs (in the untested code sections) and avert unfavorable results. Hence, higher code coverage ensures greater certainty of near zero surprises in production.

Note: To conduct code coverage analysis, you need to create a test project and add appropriate test cases (written to cover all sections of code) to it. .

Why Code Coverage

Code coverage is conducted for the following purposes:

1. Identify sections of unused code
2. Ensure that the entire code is covered by test cases
3. Identify functional bugs in sections of code that were not executed
4. Save memory by deleting sections of unused code

The process of writing test cases and reviewing the amount of code covered by test cases should be carried out during unit and integration testing. You can then add more test cases to test sections of code, which were not executed during testing, on the basis of the code coverage data.

Identifying unused objects in functions is critical as they cause memory overheads. If these unused objects are global, the memory overhead is higher as they live on till the application executes. In .NET, memory overhead caused by uncalled functions is not significant as the functions are JIT compiled.

Note: Allocation and de-allocation of memory is handled by the Garbage Collector (GC). Each time it runs, the application is put on hold, because GC runs on a high priority thread. Allocation and de-allocation of unwanted objects can cause GC to kick in frequently, impairing application performance.



How to Conduct Code Coverage

The following lists the various approaches used to conduct code coverage:

1. Treat Warnings as Errors

This approach uses a compiler to identify unused code during code compilation. During code compilation, you can set the build configuration in VS 2005 to treat all warnings as errors, as follows:

- a. In the **Project Properties** window, select **Build** tab.
- b. In the **Treat Warnings as Errors** group of the **Build** tab, enable the **All** radio button. Note that the default selection is set to **None**.

2. Run the tool via VS IDE

Code coverage from the VS IDE is integrated with the unit testing framework. You must create unit test scripts first and then execute them to get the code coverage matrix. For details on how to run the tool from VS IDE, refer to <http://msdn2.microsoft.com/en-us/vstudio/aa718857.aspx>

3. Run the Tool via Command Line

Running the code coverage tool from the command line is comparatively faster. Post running the tool run the application to perform random testing.

Note: Random testing does not usually provide high code coverage since you may miss out on testing sections of the application code. Random testing is good to catch functional bugs though.

Refer to John Cunningham's blog http://blogs.msdn.com/ms_joc/articles/406608.aspx to see how to use the command line tool.

On completion of the testing process, a report (*.coverage file) is generated that can be viewed through VS 2005. The report indicates the percentage of lines covered by test cases. If the code coverage report is generated through the Test Wizard (part of testing framework), the following additional information is available:

- a. Code lines that were not covered during testing
- b. Sections of code that were skipped during testing



The following figure illustrates a sample code coverage report:

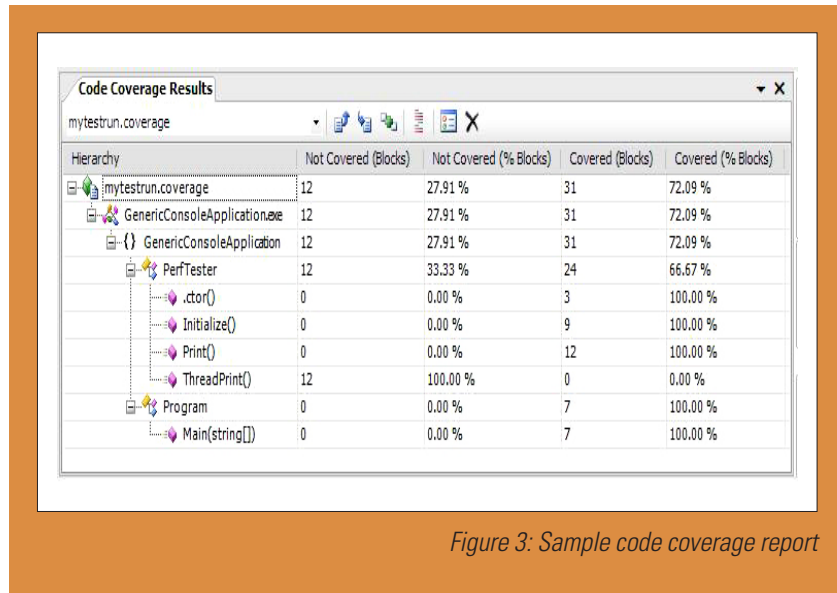


Figure 3: Sample code coverage report

For additional information on code coverage, refer to this FAQ - <http://msdn2.microsoft.com/en-us/vstudio/aa718912.aspx>

Note: Achieving 100% code coverage may not be possible due to error handling code (that doesn't execute unless there is a specific error) besides initializing and exiting routines (some of these may not work during your code coverage testing).

There are some interesting new enhancements in VS 2008 (code named Orcas, the next version of VS 2005) that will enable developers and testers to perform code coverage far more effectively. A short description of these features is covered in [The Future VS](#) section below.



Perform Function Time Profiling

Recently, Infosys conducted a short quiz and asked participants to identify methods to improve the performance of the code given to them. For more information on the code refer to the [Appendix](#)

Some of the aspects identified to improve performance were:

- a. Use typed collections instead of un-typed ArrayList
- b. Use string builders instead of string concatenation
- c. Use the for loop instead of foreach

Though these points make sense, they may not always have significant impact on the performance. In the quiz, an API call being made was taking 97% of the execution time. So, clearly, the impact of any changes to other parts of the code would be imperceptible.

This observation is in line with Pareto's famous 80-20 rule, which in this case can be stated as "80% of performance issues occur in 20% of the code". To identify that 20% of the code, you must carry out a function time profiling of the code.

Function time profiling essentially helps you analyze the time spent in each function during code execution. Using this analysis you can identify sections which, when addressed, can have a significant impact on performance.

Coming back to the quiz example, after profiling the code we identified the API that was taking 97% of the execution time and focused on resolving it. We decided to use multi-threading to parallelize the work. Then we compared the improvements after changes identified by the participants' vis-à-vis the multi-threading approach. The comparison revealed that the former yielded negligible improvements to the tune of 0.5 % or 1 %, while the latter paid off with a significant 25% benefit.

Such an insight, gained only by function time profiling and not through static code analysis, bears out the earlier recommendation against attempting to correct all errors pointed out by code analysis.

Note:

About Multi-threading

Keep in mind the following about multi-threading:

- a. Multi-threading adds overheads in the case of thread synchronization (more prominent on single core machines). Using threads excessively can actually have a negative impact since a large amount of time is wasted in thread management and context switching between threads.
- b. If the code contains several locks and sync blocks, it may not really be suitable for multi-threading. For multi-threading, the ideal case would be where each



thread executes in parallel with no dependency on another thread.

- c. Another important aspect that needs to be addressed when using multi-threading is the decision to use either .NET's thread pool or to create and use regular threads. Since .NET thread pool threads are shared resources within a process, they are good for short duration operations. A thumb rule for this could be to use them for operations that complete within one time slice or a maximum of two. Anything that takes longer should not be performed using thread pool threads. A time slice is the time an operating system allocates to execute each thread, before switching to another thread.
- d. Use the new APIs of .NET framework 2.0 (ThreadPool.SetMaxThreads and ThreadPool.SetMinThreads) judiciously and after thorough testing to ensure that there is no adverse effect due to change in thread pool size. Refer to this blog for additional details on these APIs - http://infosysblogs.com/microsoft/2007/04/how_many_threads_have_i_got.html

Multi-core is the way the industry is now moving. It will no longer be possible to depend on high processor speed for a performance boost to applications. Instead, appropriate usage of threads is fast emerging as an important way to boost the performance of code.

Tools for Function Time Profiling

The following two tools can be used for function time profiling:

- a. Performance Wizard
- b. CLR Profiler

Performance Wizard

Performance wizard of VS 2005 comes in handy for function time profile testing. It provides two options for collecting profile information – Sampling and Instrumentation. It is advisable to use the Instrumentation option in our projects.

Note: Currently you can view the reports generated by Performance Wizard only through VS2005.

Do the following to view reports generated by the Performance wizard from VS 2005:

1. In VS 2005, run the Performance Wizard. The wizard is self-explanatory.
2. Use the wizard to create a new performance test session and add appropriate properties (like [Instrumentation](#) profile)
3. Execute the application just as you would while random testing
4. Close the application. This creates the performance test report (*.VSP file). View the report in VS 2005 by opening it.

Refer to the FAQ on Profiler tools (<http://msdn2.microsoft.com/en-us/vstudio/aa718929.aspx>) to gain an insight into the tool. It also has additional links that help gain a good understanding of the tool and provide guidelines on using it effectively.

The following figure displays the various sections of the report (Performance Report Summary) generated by Performance Wizard:

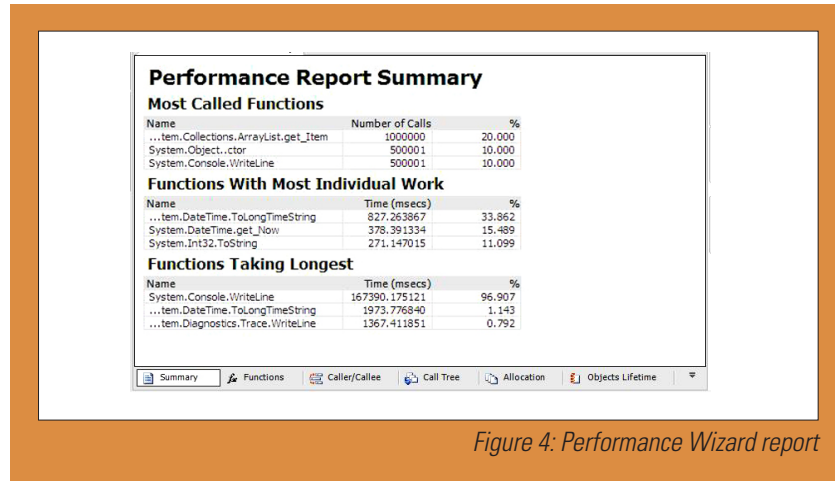


Figure 4: Performance Wizard report

The report window contains various tabs shown in the figure above. The following table describes the functionality of these tabs:

TAB	FUNCTION
SUMMARY	<p>Provides a summary of key metrics that include:</p> <ul style="list-style-type: none"> • Functions that were invoked the maximum number of times • Functions that took the longest time to execute and hence should be the focus of performance tuning • Functions that did most of the work
FUNCTIONS	<p>Provides statistics on the time spent on each function. Functions are grouped according to their parent assemblies. The attributes listed for each function include:</p> <ol style="list-style-type: none"> Number of calls – The number of times the function was invoked Elapsed exclusive time (ms) – Time spent on this function Application exclusive time (ms) – Time spent on the function excluding kernel time Elapsed inclusive time (ms) – Time spent on this function along with functions invoked from within <p>Application inclusive time (ms) – Time spent on this function and its children excluding kernel time</p>
CALLER/CALLEE	<p>Displays the same time specific details provided by the Functions tab, but from a different perspective. It includes a representation of functions that provide information such as the name of the function that invoked a particular function</p>
CALL TREE	<p>Provides yet another perspective to the same data. This essentially shows a call graph.</p>
ALLOCATION	<p>Provides details on the types allocated by the code along with information about which function calls cause the allocations. It provides details on instances, bytes allocated and the percentage of total allocations. Allocation is useful in finding out how objects are allocated and what needs to be tweaked.</p>
OBJECTS LIFETIME	<p>Displays the life time details of objects created along with Gen 0, Gen 1, Gen 2 and large object heap instances. It is important from the garbage collection perspective.</p>



Note:

- The **Functions**, **Caller/Callee** and **Call Tree** tabs help understand the time taken, source and areas of improvement of functions.
- Do not run the tool on a live application, as it slows the application.
- The Allocation and Objects Lifetime reports are available only if **Collect .NET object allocation information** and **Also collect .NET object life time information** options are selected. By default, these options are unselected.
- A command line version of the tool is available, but to analyze the report, you need VS 2005. More details on the command line version can be found at <http://msdn2.microsoft.com/en-us/teamsystem/aa718871.aspx>

CLR Profiler

Another tool that is similar to VS 2005 Performance wizard is the CLR Profiler. The CLR Profiler in most cases depicts information graphically. It provides the following elementary statistics on how functions are invoked and objects allocated:

1. Overall summary	2. Histogram of allocated types
3. Histogram of relocated types	4. Histogram of finalized types
5. Histogram of critical finalized types	6. Objects by address
7. Histogram by age	8. Allocation graph
9. Assembly call graph	10. Function call graph
11. Module graph	12. Class graph
13. Heap graph	14. Timeline
15. Class tree	

Use this information to tune the code further. The following figures show sample outputs of CLR profiler:

To view all functions that were invoked when executing the application, use the Call Tree report option of CLR Profiler. While this is an exhaustive report, it can take a long time to display if the amount of information collected is large. Links for downloading the CLR Profiler and a video on its usage, are provided in the [Additional References](#) section.

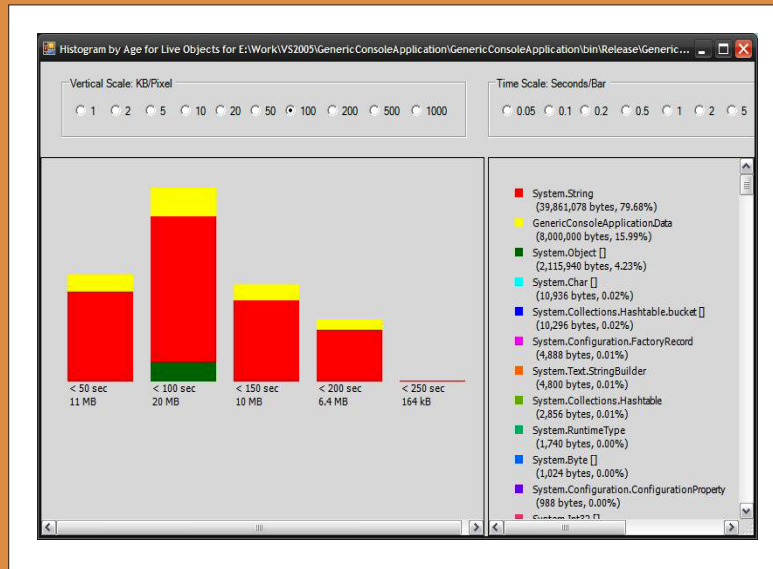


Figure 5: Histogram by Age - CLR Profiler

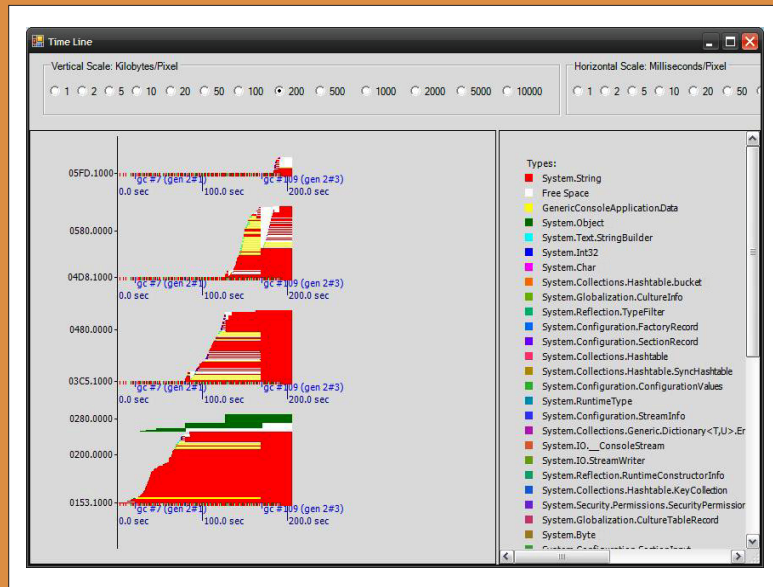


Figure 6: Time Line graph - CLR Profiler

Some other tools that you can use are:

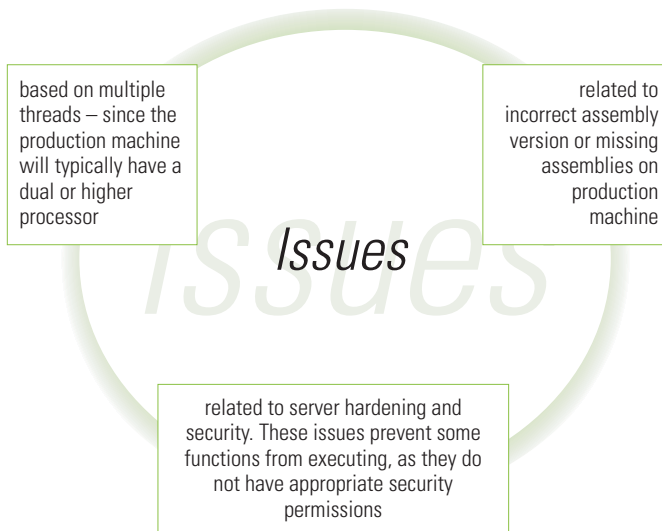
1. Intel's VTune - <http://www.intel.com/cd/software/products/asmo-na/eng/vtune/vpa/219898.htm>
2. Compuware's DevPartner Studio - <http://www.compuware.com/products/devpartner/studio.htm>

4

S T E P

Test on Production Hardware Replica

While the above steps - Code Analysis, Code Coverage Test, and Function Time Profiling - can be performed on any machine, it may not help in identifying and solving all issues. Some issues such as the following are identified only on production hardware:



Using hardware that matches the production hardware may not always be possible, but a close replica can serve as a good test bed for the applications. Rigorous testing with varying loads is critical to identify bottlenecks in the applications. Appropriate code instrumentation and capturing appropriate performance counters are essential for identifying and isolating hot spots in the code.

Most multi-thread issues surface only when there is a significant load on the application or you run the application on a multi-CPU machine. High loads can cause race conditions or deadlocks. Hence, if you use multiple threads in your application, it is imperative to run tests on a multi-CPU/multi-core machine.

Apart from the points discussed in the various steps above, general information on performance aspects like use view state judiciously, caching for ASP.NET applications, query optimization and indexing for database operations are useful while tuning your code. There are a numerous articles on MSDN and other sites on how to write high-performance code. A few sources are:



- | | |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S | 1. 10 Tips for Writing High-Performance Web Applications - http://msdn.microsoft.com/msdnmag/issues/05/01/ASPNETPerformance |
| E | 2. Taming the CLR: How to Write Real-Time Managed Code - http://blogs.msdn.com/ricom/archive/2006/08/22/713396.aspx |
| C | 3. Improving String Handling Performance in .NET Framework Applications - http://msdn2.microsoft.com/en-us/library/aa302329.aspx |
| R | 4. Performance Tips and Tricks in .NET Applications - http://msdn2.microsoft.com/en-us/library/ms973839.aspx |
| U | 5. Performance Considerations for Run-Time Technologies in the .NET Framework - http://msdn2.microsoft.com/en-us/library/ms973838.aspx |
| O | 6. Base Class Library Performance Tips and Tricks - http://msdn.microsoft.com/msdnmag/issues/06/01/CLRInsideOut/ |
| S | 7. Improving .NET Application Performance and Scalability – Guide - http://msdn2.microsoft.com/en-us/library/ms998530.aspx |

The above list is only indicative of the enormous resources available on this topic.

Finally, it is also critical to clearly understand and work toward specific performance requirements. Performance tuning can be an endless process but the benefits incurred with each change keep diminishing (in tune with the Law of Diminishing Returns) and it is important to know when to stop.

After every performance testing activity, the vital task of documenting the findings ensures that the knowledge is preserved for new teams that may work on the code in future. Without exhaustive documentation, future teams would be clueless about specific optimizations and may even undo some of the changes causing performance degradation.



THE FUTURE VS

The next version of VS called VS 2008 is expected to release by the end of February 2008. There are quite a few new improvements expected in code profiling and performance testing. A few key features in VS 2008 are as follows:

1. Ability to identify complex and error prone code with cyclomatic complexity calculations
2. Ability to baseline a particular performance run and compare it with other runs. This helps in finding out whether changes to the code improved the performance
3. Hot pathing to help identify code paths that cause maximum performance issues

Check this blog for insights into the upcoming version of VS http://infosysblogs.com/microsoft/2007/04/vsts_orcas_beta_1_profiler_com.html.

For a detailed listing of the new features in Orcas and Rosario, refer to <http://msdn2.microsoft.com/en-us/teamssystem/bb407307.aspx>.

C O N C L U S I O N

High performance code is no fluke. The key to building a high performance application is not limited to writing good code. Careful planning, a clear understanding of requirements and choosing the right approach are all crucial for extracting the best performance from code. The iterative process of performance tuning with its clearly defined end state helps achieve this objective.

Additional References

1. Writing Faster Managed Code: Know What Things Cost: <http://msdn2.microsoft.com/en-us/library/ms973852.aspx>
2. Writing High-Performance Managed Applications: A Primer - <http://msdn2.microsoft.com/en-us/library/ms973858.aspx>
3. PnP Performance Testing Guidance - <http://www.codeplex.com/PerfTesting>
4. CLR Profiler
 - a. For .NET Framework 2.0
<http://www.microsoft.com/downloads/details.aspx?familyid=A362781C-3870-43BE-8926-862B40AA0CD0&displaylang=en>
 - b. Gregor Noriskin's video on MSDN TV
<http://msdn.microsoft.com/msdntv/episode.aspx?xml=episodes/en/20030729clrgn/manifest.xml>
 - c. Introduction to CLR Profiler by Peter Sollich
<http://www.microsoft.com/downloads/details.aspx?familyid=CD5AA57C-9CD1-45AB-AA4B-8DC586D30938&displaylang=en>

We conducted a quiz (refer Step 3 Perform Function Time Profiling) in which participants were asked to optimize the performance of the following code:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Collections;

namespace GenericConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            PerfTester pf = new PerfTester();
            pf.Initialize();
            pf.Print();
            Console.WriteLine("Done");
        }
    }

    internal class PerfTester
    {
        private ArrayList list = new ArrayList();

        public void Initialize()
        {
            for (int i = 0; i < size; i++)
            {
                Data d = new Data();
                d.s1 = i.ToString();
                d.s2 = DateTime.Now.ToLongTimeString();
                list.Add(d);
            }
        }

        public void Print()
        {
            foreach (object o in list)
            {
                string str = ((Data)o).s1;
                str += "," + ((Data)o).s2;

                Console.WriteLine(str);
            }
        }
    }

    internal class Data
    {
        public string s1;
        public string s2;
    }
}
```

About the Author:

Atul Gupta, a Principal Architect at Infosys' Microsoft Technology Center, has over 12 years of experience in Software Industry. He consults on latest Microsoft technologies. Earlier he focus was BizTalk Server 2006. He is now exploring Windows Presentation Foundation (WPF) of .NET framework 3.0. He blogs regularly at <http://blogs.infosys.com/microsoft> on latest technologies. He is also Microsoft MVP Solutions Architect (<https://mvp.support.microsoft.com/default.aspx/profile/atul.gupta>). He can be reached at AtulG@Infosys.com.

Questions? Opinions?

Discuss Visual Studio 2005 at
<http://blogs.infosys.com/microsoft>

Infosys Technologies Ltd. (NASDAQ: INFY) defines, designs and delivers IT-enabled business solutions that help Global 2000 companies win in a flat world. These solutions focus on providing strategic differentiation and operational superiority to clients. Infosys creates these solutions for its clients by leveraging its domain and business expertise along with a complete range of services.

With Infosys, clients are assured of a transparent business partner, world-class processes, speed of execution and the power to stretch their IT budget by leveraging the Global Delivery Model that Infosys pioneered.

Infosys[®]

POWERED BY INTELLECT
DRIVEN BY VALUES

For more information, contact catalyticit@infosys.com

www.infosys.com

© 2007 Infosys Technologies Limited, Bangalore, India. Infosys believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. Infosys acknowledges the proprietary rights of the trademarks and product names of other companies mentioned in this document.