Czech Technical University, Prague
Faculty of Electrical Engineering

Master's Thesis

Performance Profiling for .NET Platform

Bc. Jan Vratislav

Supervisor: Ing. Miroslav Uller

Study Program: Electrical Engineering and Information Technology

Study Branch: Computer Science and Engineering

January 2012

Declaration

I hereby declare that I have completed this master thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zakona c. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, January 3rd, 2012 ....

Acknowledgement

I thank to Ing. Miroslav Uller for accepting and supporting my thesis.

I thank to my closest and my friends for their support.

Abstract

The so-called cascading undo command has been introduced by Aaron Cass and Chris Fernandes . This new approach to the undo command overcomes the weakness of the linear undo command. It allows undoing an arbitrary action from the history while watching the dependencies among the actions. However, there is not a visualization of cascading undo yet. Thus, in this thesis we discuss, introduce, develop, and evaluate several visualizations for cascading undo. Unlike the linear undo visualizations, cascading undo visualizations have to deal with dependencies among user actions. We believe that an overview of the dependencies should be presented to a user before committing and undo command. The visualizations we proposed are flexible enough to reflect the possible complexity of the user actions and their dependencies.

Abstrakt

Aaron Cass a Chris Fernandes představili takzvaný kaskádový příkaz zpět. Tento nový způsob příkazu zpět překonává slabiny všudypřítomného lineárního příkazu zpět. Umožňuje totiž zrušit libovolnou akci z historie akcí dokumentu. Při tom bere v potaz závislosti mezi těmito akcemi. Nicméně nikdo ještě nevyvinul vizualizaci pro kaskádový příkaz zpět. V této práci diskutujeme, představujeme, vyvíjíme a hodnotíme několik vizualizací kaskádového příkazu zpět. Na rozdíl od vizualizace lineárního příkazu zpět musí kaskádový příkaz zpět počítat se vzájemnými závislostmi provedených akcí. Věříme, že uživatelé by měli mít přehled o těchto závislostech ještě před jejich vlastním odebráním. Námi navržené vizualizace jsou flexibilní natolik, aby zvládly zobrazit komplexitu uživatelských akcí a závislostí mezi nimi.

Contents

List of Figures

Chapter 1

Introduction

According the Pareto's law (also known as 80/20 rules) 80 percent of the results comes from 20 percent of the effort [Koc99]. This law is applicalbe to software development. 80 percent of all end users generally use only 20 percent of a software application's features. Microsoft reported that 80 percent of errors and crashes in Windows and Office are debited to 20 percent of bugs [Roo02].

The same principle applies to software performace. 80 percent of time is spent in 20 percent of code. Some argue that it is even more, 90/10. So, investing programmers' effor to the 20 percent of code may have great effect on the overall speed - if that 20 percent of code be discovered.

Programmers are not particularly successful at guessing which part of the code is crutial for the perfomance [McC04]. Therefore profilers help to automate the search of bottleneck and hotspots in applications and their source code and provide valuable metrics, such as execution times of specific part of code or memory usage of a given object. .NET programs' performance profiling ist the main area of this thesis.

The performace profiling dynamically analysises execution behaviour of a program. It tracks various runtime related data as frequency and duration of function calls. Analysis and visualition of the gathered data provides useful hint on the program's code runtime characterics and helps during optimalization and exploration of the program.

Profiling can be achived by various means as e.g. by inserting tracing code into either the source code or the binary executalbe of the program or by runtime sampling of thread call stacks of the program or by listening to events invoked by a program's runtime engine.

Each of aforementioned approaches differs in overhead imposed to the program and in kind, precision and granuality of the gathered data.

In the world of .NET performace profiling exist already few full-fledged solutions targeting almost all .NET platforms from desktop to Windows Phone applications. However, they are mostly commercial and do not provide deep integration into development tools.

In this thesis, we will to introduce a development-time .NET profiler offering various profiling methods and allowing direct interaction directly from the Microsoft Visual Studio 2010.

Chapter 2

Overview of Performance Profilers

In this chapter we present various characteristics and mechanisms of performance profilers. We will overview some contemporarily used profilers targeting .NET platform.

2.1 Profiling modes

An application can be profiled in several ways that differ in the results' precision, profiling overhead.

2.1.1 Sampling profiling

In this mode, the profiler stops periodically every profilee's thread and inspects method frames on its call stacks. The output snapshot is not an exact representation of the runtime conditions rather a statistical approximation, but the profiling overhead is very low and the profilee runs almost in full speed with minimum side effects, such as additional memory allocations, cache faults and context switches.

The profiling overhead does not depend on the number of method called by the profilee as in other modes. The introduced error of the results is equal to half of the sampling period.

The profiler cannot count the number of method calls and the method duration are computed purely based on gathered statistics.

No source code or binary alternations in required, the runtime ability to stop execution to do a call stack snapshot or alternatively an interrupt instruction can by used.

2.1.2 Tracing profiling

This way of profiling relies on a runtime environment or some kind of hardware notification when a method is entered and left and leads to accurate results, providing exact count of method calls and their durations, however, with some added overhead. In addition, the overhead increases with the method calls count and slows down the profilee.

As with the sampling profiling, there are no changes to code or binaries required. The profiler has to only register callback methods for the entry/leave notifications with the runtime or the hardware profiling infrastructure.

2.1.3 Instrumentation profiling

The profilee code or binaries are modified by injection of arbitrary code or instructions in order to collect profiling data. This approach can have similar effects and results as the tracing profiling, but offers a lot more choice of what and how to profile. There are chances of alternation the original profilee behaviour or even introducing bugs.

Instrumentation can be perform on every level of the software live-time (source, compilation, binary, runtime) and can be both manual, performed by a developer, as well as automatic, done by a compiler or a runtime environment.

2.2 Granularity of profiling

A profiler can measure profiling result on different levels of a program. Not every profiling approach can achieve any granularity due to its limitations. Speed of the profilee is always trade-off between granularity and accuracy.

2.2.1 Line-by-line granularity

Line-by-line profiling is the most accurate method and most demanding. Every program statement is measured and analysed. It brings precise result, however, the additional burden can alter the program's behaviour.

Such fine granularity can be achieved only by the instrumentation profiling.

2.2.2 Method granularity

Only information regarding an entire method run are collected. The lower overhead can ease the profilee and still provide very informative results.

This level of granularity is the only option for the sampling profilers and most cases the tracing profilers. The instrumentation profilers can be easily converted from the line-by-line to the method granularity.

2.2.3 Selective granularity

The profiling process can be filtered with combined levels of granularity, where some parts of code are monitored on the line-by-line and others only on the method manners or not at all in order to capture desired statistic with lowest possible overhead and highest possible accuracy.

	granularity			
profiling mode	method	line-by-line	overhead	accuracy
sampling	yes	no	very low	moderate
tracing	yes	no	high	high
instrumentation	yes	yes	very high	very high

Figure 2.1: Profiling modes comparison

2.3 Profiling results

During the process of profiling various kinds of data can be collected. For some application, it is sufficient to count the method hit count for others is a detailed call stack analysis required. Again, the amount of result information places overhead on the profilee, in this case primary on the memory (caches, pages...) and secondary on the CPU.

2.3.1 Time measurement

There are several different kinds of the program time measurement a profiler can provide. Not every profiling mode can provide every single measurement. As usually, the measurement accuracy is counterweighted with the computational complexity.

Wall time

The wall time measurement starts when a thread enters a method and ends when the thread leaves the method. The resulting time does not reflect if the method does useful computation or is in either a wait, a sleep or a join mode.

User and kernel time

In comparison with the wall time, the user and kernel time measurement counts solely time spent executing a method exclusive the time spent by waiting, sleeping or joining.

Every profiling mode is capable of both the wall time as well as the user and kernel time measurement, either by statistically distributing the program execution time over the method hit counts or by reading CPU registers or system performance counters.

2.3.2 Call trees

When relations among methods calls allows the profiler to reconstruct a call tree. The call tree reveals hit count of functions and, more importantly, what function calls which function. This insight helps to find hot spots and understand the runtime conditions. Additional function parameter analysis is also possible, but in cost of higher CPU and memory overhead. A example of a call tree is shown on the figure 2.2.

Some profilers do not track calls hierarchy and they only record function hit count and duration. It is referred by the term flat call tree.

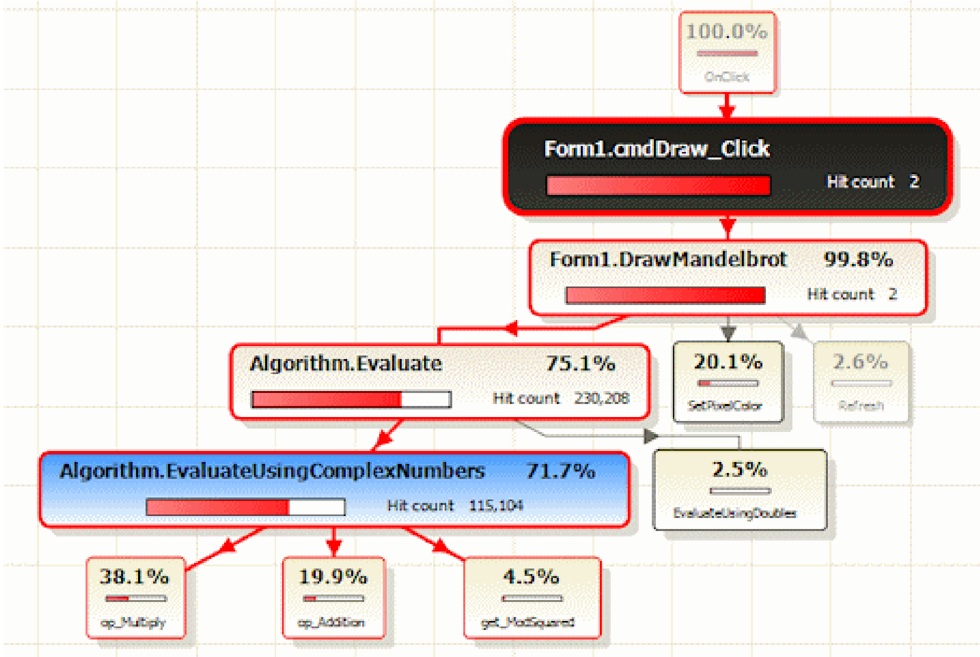

Figure 2.2: Red-Gate ANTS Profiler call graph

2.4 Available .NET performance profilers

On the market, there is many various options in the field of .NET performance profiling. The commercial solutions offer many features and integration with others tools, however, not with Visual Studio in most cases. There are also a few open source alternatives.

Unfortunately, we could not carry out deeper performance and features assessment of available profilers, since we do not have required financial and time resources for this challenging and but very interesting undertaking.

Commercial solutions

2.4.1 JetBrains dotTrace

dotTrace profiles .NET Framework 1.0 to 4.0, Silverlight 4, or .NET Compact Framework 3.5. It offers partial integration with the Visual Studio and others JetBrains tools. All the profiling modes as presented and the profiling can run remotely.

www.jetbrains.org

2.4.2 Redgate ANTS Performance Profiler

This tool targets similar set of .NET applications. In addition it offers SQL and I/O profiling, live results, all the profiling modes and time schemas.

It offers some integration in the Visual Studio and allows to see code directly from the profiling tool.

www.red-gate.com

2.4.3 EQATEC Profiler

There is no doubt that this profiler offers the widest targeting platforms options. It can be used to profiler virtually anything in the ".NET world". It uses some kind of instrumentation and thus the choice of the profiling mode is restricted, however it is configurable. A binary has to be modified before a profiling session.

There is no integration with the Visual Studio whatsoever and it seems that the source code result overview cannot be displayed either.

www.eqatec.com

2.4.4 GlowCode

GlowCode is a performance and memory profiler for Windows and .NET programmers who develop applications with C++ or any .NET Framework language. GlowCode helps to detect memory leaks and resource flaws, isolate performance bottlenecks, profile and tune code, trace real-time program execution, ensure code coverage, isolate boxing errors, identify excessive memory usage, and find hyperactive and loitering objects. For native, managed, and mixed code. [ES11]

We were unable to find anything about the Visual Studio integration. So we assume that it is not supported.

www.glowcode.com

2.4.5 Visual Studio Profiler

This tool is integrated to the Visual Studio and supports the native and managed code profiling. It has the sampling and instrumenting profiling modes. It is shipped only with higher editions of the Visual Studio

2.4.6 Others

There is even more very comparable profiling suites offering very similar functionality as aforementioned solutions as the Telerik JustTrace, the SpeedTrace Pro and maybe even other that we have not discovered.

It is a challenging task to pick the right solution for one's needs with the right licensing options.

Open source solutions

The open source choice is not as vast as the commercial, so far offering only two solutions.

2.4.7 SlimTune

SlimTune is a free profiler that offers advanced features as remote profiling 32-bit and 64-bit profiling, live results. There is very little information available and user has to dive into the source code to find out about its inner mechanisms.

code.google.com/p/slimtune

2.4.8 Prof-It for C#

A unique way of the profiling introduces the Prof-It for C# profiler. It is a line-by-line instrumentation profiler with its own source viewer allowing import and profiling of Visual Studio projects. It present result as a overlay over the source code and as a list of methods, blocks and classes.

Unfortunately, this project does not seem to be actively developed.

dotnet.jku.at/projects/Prof-It

Summary

In this we have look at and explained the sampling, the tracing and the instrumenting profiling modes. Then we focused on the profiling results and their granularity, mainly the call trees and different kinds of time durations. In the end of the chapter a short overview of the current commercial and open-source scene was presented.

Bibliography

[ES11] Inc. Electric Software. The glowcode 8.2 website. *http://www.glowcode.com/*, 2011.

[Koc99] Richard Koch. *The 80/20 Principle: The Secret to Achieving More with Less.* Crown Business, 1999.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, 2nd Edition.* Microsoft Press, 2004.

[Roo02] Paula Rooney. Microsoft's ceo: 80-20 rule applies to bugs, not just features. *http://www.crn.com*, 2002.