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

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 performance 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.
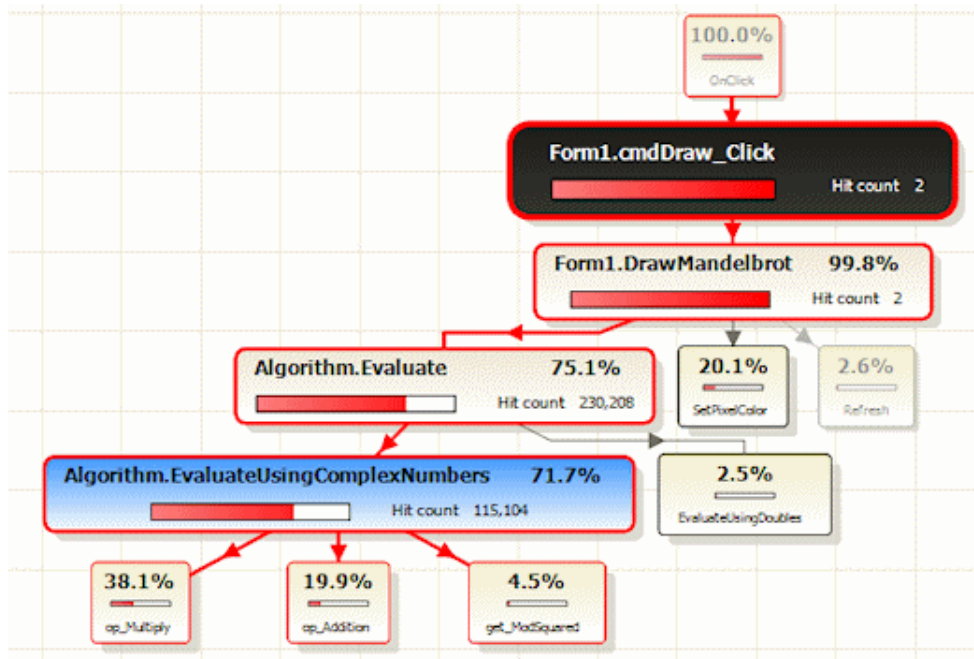
Figure 2.2: Red-Gate ANTS Profiler call graph

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 presents results as an 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 inspected durations. In the end of the chapter a short overview of the current commercial and open-source scene was presented.

Chapter 3

Performance Profiling on .NET

In this chapter, we will first briefly review some .NET framework core facilities, mainly CLR, and then explore options to the performance profiling on this platform.

3.1   .NET framework core facilities

.NET framework (of just .NET) is a software platform developed by Microsoft for developing and executing software applications. .NET consist of a runtime environment, which executes the programs, class libraries, interfaces and services. It offers to software developers ready to use solutions for standard programming tasks, so they only have to concentrate on a program logic itself.

.NET based applications do not contain machine-level dependent instructions and cannot be, therefore, directly executed by CPUs. Instead of that, they are composed from a human readable "middle" code instructions, so called Common Intermediate Language (CIL). In order to execute CIL, it has to be first compiled into processor specific machine-level instructions by the JIT compiler (Just-in-time compiler) that uses processor specific information during the compilation. This mechanism is comparable to Java and its bytecode.

The .NET platform is an implementation of Common Language Infrastructure standards (CLI) and forms a solid base for development and execution of programs that can be written in any programming language on any platform. The center elements of .NET are the runtime environment Common Language Runtime (CLR), the Basic Class Library (BCL) and diverse utility programs for management and settings.

Many programming languages such as C#, F#, Visual Basic and others can be translated to the CIL and then run on the CLR.

CLR provides huge amount of services and features. Besides already mentioned ones the JIT compilation also provides assemblies loading, type system, garbage collection and memory management, security system, native code interoperability, debugging, performance and profiling interfaces, threading system, managed exception handling, application domains and many others.
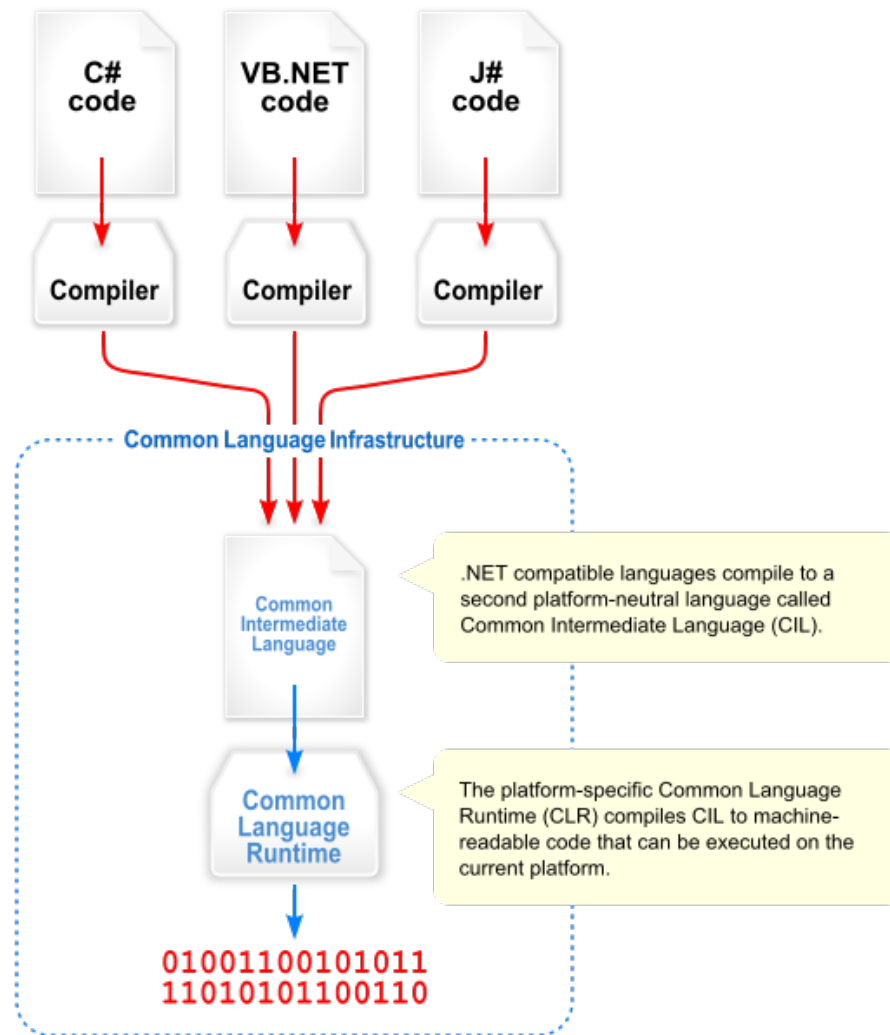
Figure 3.1: Overview of the Common Language Infrastructure [Wik11]

3.2 Profiling modes in .NET

All the profiling modes (sampling, tracing, instrumentation), chapter 2, are feasible in the .NET environment. There is even more ways how to achieve some of them, however, with very different implementation complexity.

3.2.1 Sampling mode

Due to the JIT compilation, real address of a function remains unknown till the point of the JIT compilation of the function, which might not even occur if the function is not on some program's execution path. Therefore, there is not a direct way how to acquire the function and its memory address mapping without digging into the CLR runtime structures. Luckily, the Profiling API answers this problems and let a programmer sample all managed call stacks and read metadate regarding the methods.

3.2.2 Tracing profiler

As stated in the introduction in the chapter 2, the tracing profiling must be supported by the runtime engine or the targeting hardware. The Profiling API offers a very elegant way to implement this profiling strategy and to trace the CLR activity.

3.2.3 Instrumentation profiling

This mode is the most difficult, nevertheless, with the most profiling possibilities.

The first possibility how to create a .net instrumenting profiler is by an injection of the profiling source code right into the source code before a compilation from a high level programming language to the CIL. Various features of the higher level language have to be taken in the account during this process, e.g. lambda methods and closures in C#. Obvious limitation is only one target programming language.

The second possibility is similar to the first and it only goes one level down, to manipulate CIL of already compiled .NET assemblies or modules. This approach targets all every .NET program regardless of the programming language. However, certain problems have to be solved, e.g. the target assembly signing.

The third possibility is to use the JIT compilation notifications of the profiling API. The original CIL can be modified on before the JIT compilation.

It is apparent how big the role the profiling API in the profiling of .NET applications plays. It is a great starting point for every profiling mode implementation.

3.3 Profiling API

This section is based on articles, documentation and examples provided on the Microsoft Developer Network (MSDN) [Mic11]. The provided information is valid for the version of the profiling API that comes with the .NET 4.0.
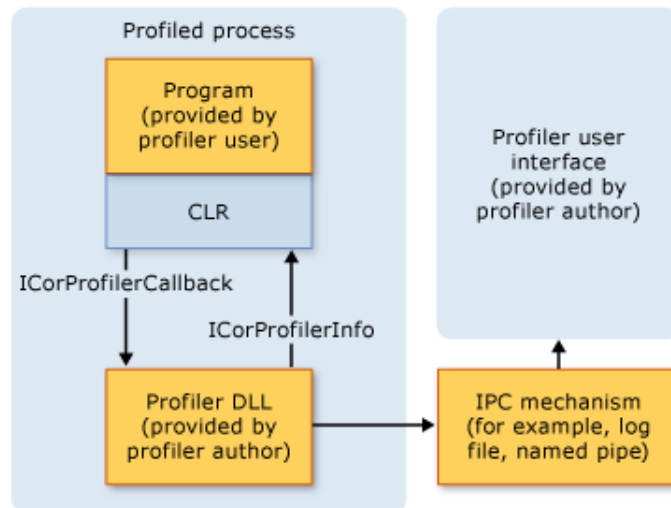
Figure 3.2: Profiling architecture [Mic11]

Profiling a .NET application is not as straightforward as profiling a conventional application compiled into the machine code. The main reason are aforementioned CLR features that the conventional profiling methods cannot properly identify and thus allow the profiling. The profiling API provides a way how to acquired the profiling data for a managed application with minimum effect on the performance of the CLR and the profiled application.

The purpose of the profiling API is not to be only a profiling tool, as its name suggests, but it is a versatile diagnostic tool for the .NET platform that can be used in many program analysis scenarios, e.g. a code coverage utility for unit testing.

To make use of the profiling API, as depicted on the figure 3.2, a profiler's native code DLL containing profiling custom logic is loaded to the profiled application's process. The profiler DLL implements the *ICorProfilerCallback* interface and creates a in process COM server . The CLR calls methods in that interface to notify profiler of one of many runtime events of the profiled process. The profiler can query the state of the profiled application and the events by calling back into CLR using methods of *ICorProfilerInfo* interface.

It is important to keep the profiler DLL as simple as possible. Only the data monitoring part of the profiler should be running in the profiler process. The rest of the profiler, analysis and UI, should happen in an independent process.

Unfortunately, the profiler DLL cannot be written in managed .NET code. It actually makes sense. It the interfaces were implemented using managed code, it would be self triggering the CLR notification and would eventually end up in never ending recursion or a deadlock. For example, imagine that you register a method entered notification handler and you call a managed method within the handler. You would get notified of that.

After the COM implementation of the interfaces is finished, the CLR needs to be aware of the profiling DLL and forced to load it and call its methods. Simply enough, this is accomplished by setting environment variables (*COR_ENABLE_PROFILING*, *COR_PROFILER, COR_PROFILER_PATH*). No XML setting, no registry entries. Be

careful with that! If you changed the machine wide environmental variables then every
.NET process would be profiled. The profiler can even be attached in the runtime.

3.3.1 Supported notifications

The CLR notification can be divided into following groups as listed in [Mic11].

1. CLR startup and shutdown events.

2. Application domain creation and shutdown events.

3. Assembly loading and unloading events.

4. Module loading and unloading events.

5. COM vtable creation and destruction events.

6. Just-in-time (JIT) compilation and code-pitching events.

7. Class loading and unloading events.

8. Thread creation and destruction events.

9. Function entry and exit events.

10. Exceptions.

11. Transitions between managed and unmanaged code execution.

12. Transitions between different runtime contexts.

13. Information about runtime suspensions.

14. Information about the runtime memory heap and garbage collection activity.

The implementation of the *ICorProfilerCallback* interface is mandatory to use the pro-
filing API. The developer of the profiler has to implement all methods of the interface
and there is 80 of them! Luckily, for the majority of methods, those out of interest, it
is simply enough to return S_OK HRESULT. The others have to provide code to handle
the events and desired logic.

Most of methods of the *ICorProfilerCallback* interface receives additional data in form of
input parameters. And some methods come in pairs started-finished (*GarbageCollectionStarted-
GarbageCollectionFinished*) or enter-leave (*ExceptionSearchFunctionEnter-ExceptionSearchFunction.*

The parameters are either unsigned int (UINT) Ids or pointers to some CLR structures.
E.g. *ThreadCreated* get its threadId parameter of type ThreadID (only *typedef*ed UINT).
The Ids are opaque values, but we think they are bare pointers to the memory where the
metadate reside since they are always defined as UINT_PTR.

To get more information such as functions name, defining class, assembly, actually pretty
much everything, what is in the metadate, from the Ids, the profiling API offers the

ICorProfilerInfo interface and the *IMetadataImport* interface and system of metadata Tokens corresponding with the metadata.

The profiling API offer a lot and there is a lot more to write about. However there also some limitation to it. Among others no profiling for unmanaged code and no remote profiling.

3.4   Summary

In this chapter, the .NET framework was briefly introduced, we have outlined some .NET profiling strategies for all the profiling modes and introduced the powerful profiling API for profiling managed applications.

Chapter 4

The Visual Profiler Backend

The backend is the part of the profiler resposible for collecting the data from the profiled application and for sending them to the analitic part of the Visual Profiler frontend. There are many requirement for a profiling backend to fulfill, such as speed, mutli-threadinng, correctness, low memory consumption and others. In this chapter, we will introduce the profiling modes we chose to implement and review the inner implementation of the backend and desing decision we had to make.

4.1 Choice of the implentation strategy and the profiling modes

Based on the analysis of the profiling modes on the .NET platform in the chapter 3. We made a decision to use the profiling API as the base for our own profiler before the implementing everything from scratch. The profiling API offers a matured API with intrinsic support of all .NET features. This helps to avoid many unnecessary implementation problems that could be encountere during an implementation of a .NET profiler entirely from beginning.

In the abstract of the thesis we have committed ourself to create a method granularity profiler supporting two distinct profiling modes for the .NET.

The choice of the sampling profiling mode was straightforward because of its stochastic nature in that it completely differs from the other two modes.

On the other hand, the difference between the tracing and the intrumentation profiling is not that vast. They both measure exact results and put similar overhead on the profiling. However, the selected method granurality makes them both even more comparable from user's point of view. They both have their implementation pitfall and neither of them would be easier to progam that the other. In the end, we opted for the tracing profiling mode since it does not require any changes to target assemblies.
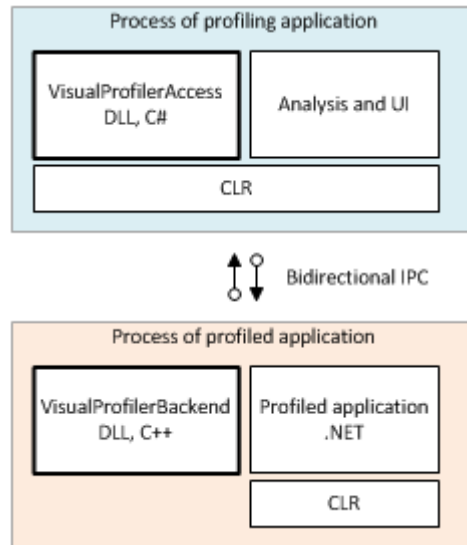
Figure 4.1: The backend's architecture

4.2   Software architecture of the backend

The backend runs in two process. The first process, written in C++ and called the Visual Profiler Backend, is the actual profiled application with the hosted profiler DLL that is loaded by the CLR and accessed by the COM interface, as described in the chapter 3. The sampling and tracing profiliers are implemented as distinct COM CoClasses and only one of them can run at a time. The CoClasses share the common code base for the profiled data collection and the interprocess communication with the managed code.

The second process is programmed in C# and called the Visual Profiler Access. It starts the profiled application, written in C++, with the desired profiling mode in a separate process, initialize the interprocess communication for transferring the profiling data and is responsible for their processing so they can be later used by an analystic and UI frontend.

The whole relation is shown on the figure 4.1.

4.2.1   The Visual Profiler Backend

In this part are the both chosen profiling mode implemented. It also contains bunch of supporting classes for handling metadata information for methods, classes, modules and assemblies, for caching the profiling intermediate results and for serializing the results and transmitting them over IPC to the Visual Profiler Access. The Active Template Library (ATL) framework from Microsoft is used to simplify programing of the Component Object Model (COM) in C++

Let us now introduce you the inner structure and rough outline.
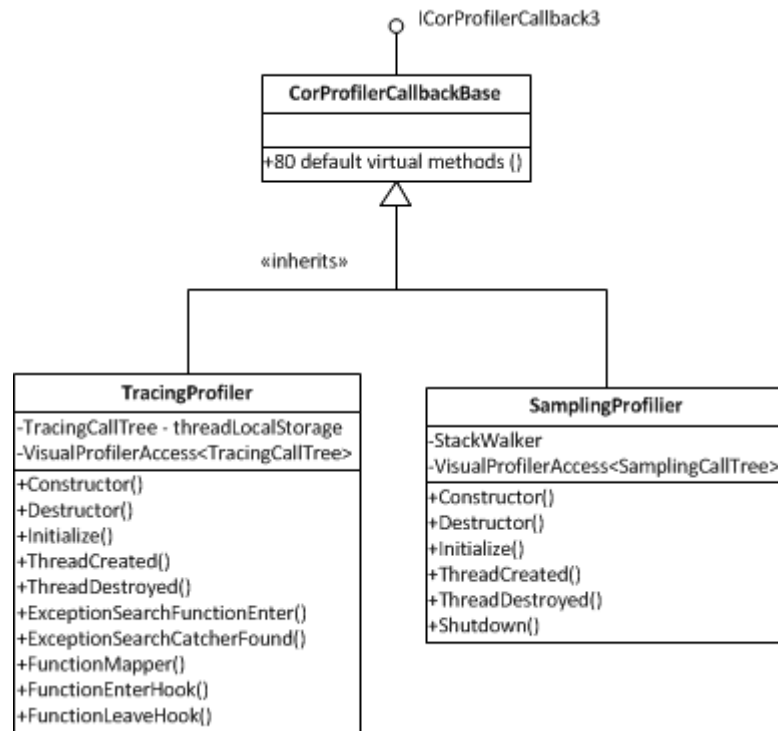
Figure 4.2: Implentations of the *ICorProfilerCallback3* interface

4.2.1.1 Implentations of the *ICorProfilerCallback3* interface

The *ICorProfilerCallback3* interface is the center point of the profiling API. With its 80 methods it is a little bit unpractical to implement it, mainly because only a handful of the method is usualy made use of. For that reason we created a default class *CorProfilerCallbackBase* as depicted on the figure 4.2. This provides empty implementation of every methods of the interface and derived classes can only override desired methods. This idea makes it easier to have clearly arranged classes, the clean code rules.

Thebothclasses, theTracingProfilerandtheSamplingProfiler, overridetheInitialize, ThreadCreatedand

The *TracingProfiler* class

This class is home for the tracing profiler mode implementation, the figure 4.2. It starts its lifetime by running the IPC with the Visual Profiler Access on a separate thread in the *Constructor()* method. After that, the *Initialize()* method is invoked by the CLR and passed a pointer to the *ICallProfilerInfo3* interface. Here registers the profiler the function ID mapper (more detail later in this subsection), the function enter/leave, the thread created/destroyed and the exception events notifications. At this point is everything set up and the profiled application starts running.

The profiler receives a notification from the CLR whenever a new managed thread is created - via the method *ThreadCreated*. The notification executes within the context of the newly created thread. The tracing profiler registers the thread id and associates with it an instance of the *TracingCallTree* class. Such an association is saved into a
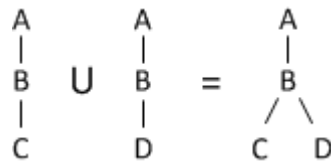
```
A         A           A
|         |           |
B   U     B     =     B
|         |          / \
C         D         C   D
```

Figure 4.3: Union of thread's call stack snapshots forming a call tree. A very simple example: A calls B, B calls C, C returns to B, B calls D, D returs to B, B return to A

hash table, represented by C++ *std::map*, under the thread id key. The reference to the *TracingCallTree* object is stored into a thread local storage in order to avoid repetitive search in the hash table by every single enter/leave notification.

During the profiling session there is a need to trace the method enter and leave notifications for every managed thread simultaneously. This is a demanding task if you take in account the every single call to and return from a managed method is augmented by overhead of this notifications.

Our first approach to store the collected profiling information was to push the data, a function ids, into an enter stack and a leave stack. Every managed had its own private enter and leave stacks. All stacks were saved by the corresponding thread ids in a common hash table. During the processin of a funtion enter or leave notification the hash table had to be searched. This created a potential race condition (inserting a new thread to hash table and reading from it at the same time). So the critical section had to be added to access the hash table.

That was fairly easy to implement and did indeed worked. However, performed not very well. Firstly, it used so much memory that after tens of seconds the application memory working set reached hunderts of mega bytes. Secondly, the syncronized hash table lookups slowed down the application. The profiled application was running approximately 50 times slower! (according to the rough duration messurements in the profiled application). It was clearly unacceptable. A better solution had to be found.

Subsequently, we realized that an union of snapshots of thread's call stacks forms a call tree as depicted on the figure 4.3.

The *TracingCallTree* class was created based on the idea. A simplified representation of the TracingCallTree class is depicted on the figure 4.4. Every tree element (class TracingCallTreeElem) contains commulative profiling results such as the number of the enter and leave notifications, the user, the kernel and the wall-clock time related data and so other auxiliary data. The object of the TracingCallTree class contains an active element pointer and a root pointer. The active element pointer points to an eleme representing a currently executing function and the root element pointer represents the top most method, e.g. the Main method.
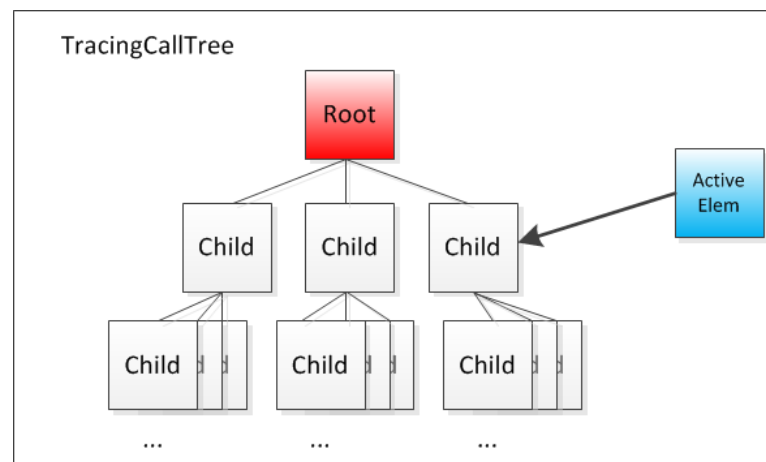
Figure 4.4: A simplified representation of the TracingCallTree class. The active element pointer points to the currently executing function.
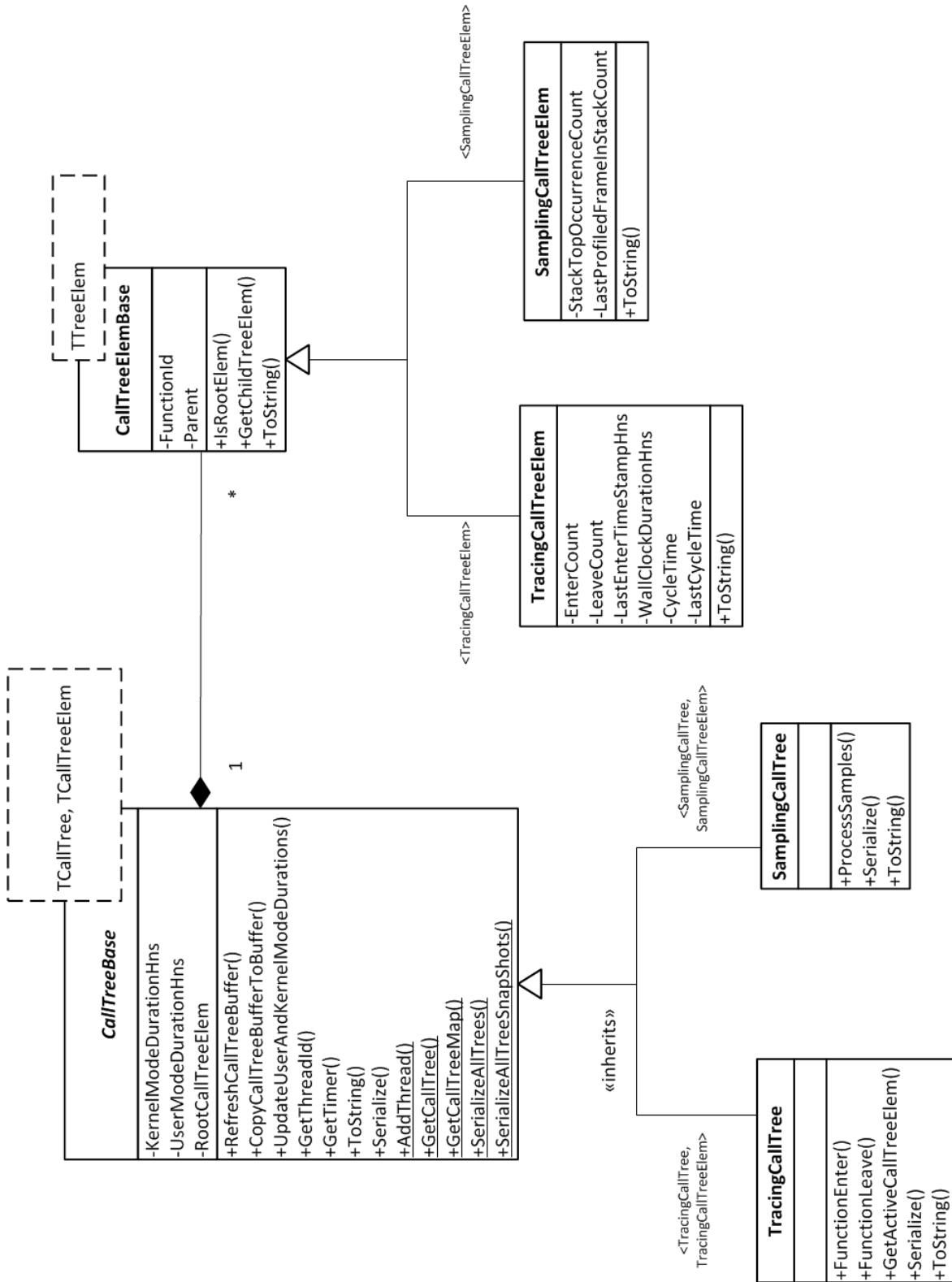
Figure 4.5: The hierarchy of the call tree classes used to trace the collected profiling data.

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.

[Mic11] Microsoft. Profiling (unmanaged api reference). `http://msdn.microsoft.com/en-us/library/ms404386.aspx`, 2011.

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

[Wik11] Wikipedia. .net framework. `http://en.wikipedia.org/wiki/File:Overview_of_the_Common_Language_Infrastructure.svg`, 2011.