

# Towards Effective Runtime Trace Generation Techniques in the .NET Framework \*

Krisztián Pócza	Mihály Biczó	Zoltán Porkoláb
Eötvös Loránd University	Eötvös Loránd University	Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Lang. and Compilers	Fac. of Informatics, Dept. of Programming Lang. and Compilers	Fac. of Informatics, Dept. of Programming Lang. and Compilers
Pázmány Péter sétány 1/c.	Pázmány Péter sétány 1/c.	Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary	H-1117, Budapest, Hungary	H-1117, Budapest, Hungary
kpocza@kpocza.net	mihaly.biczo@axelero.hu	gsd@elte.hu

## ABSTRACT

Effective runtime trace generation is vital for understanding, analyzing, and maintaining large scale applications. In this paper two cross-language trace generation methods are introduced for the .NET platform. The non-intrusive methods are based on the .NET Debugging and Profiling Infrastructure; consequently, neither additional development tools, nor the .NET Framework SDK is required to be installed on the target system. Both methods are applied to a test set of real-size executables and compared by performance and applicability.

## Keywords

Runtime trace generation, .NET, Debugger, Profiler, program slicing

## 1. INTRODUCTION

Generating and analyzing runtime traces for large scale enterprise applications is a common task to investigate the cause of arising malfunctions and accidental crashes.

Using a debugger application, examining the application log or the event log of the operating system in order to find the erroneous instructions and the variables getting incorrect values in the program can be very useful. However, there are many situations where a simple debugger fails to find the erroneous instructions and variables. One common example is when the error occur in a production environment where we are not allowed to install a development environment to detect the bug [Mar03a]. Furthermore, multithreaded applications or applications producing incorrect behavior only under heavy load often may not be debugged correctly on the development machines. What make things even

more complicated is that in the case of programs and components that run on a deployment server or a client computer incompatibility issues might also arise. Further problematic situations include cases when the deployment servers are in a Network Load Balancing (NLB) Cluster, or the isolation level on the IIS web server is too restrictive.

The most common research area where low level runtime traces are used in the academic world is dynamic program slicing [Agr91a, Bes01a, Póc05a, Tip95a, Zha03a]. The result of program slicing can be used in the industry also. The original goal of program slicing was to map mental abstractions made by programmers during debugging to a reduced set of statements in source code. With the help of program slicing programmers are able to identify bugs more precisely and at a much earlier stage.

In this article we show two different methods for generating source code statement level runtime traces for applications hosted by the Microsoft .NET Framework 2.0. In their current form our solutions are incompatible with older versions (1.0, 1.1) of the .NET Framework but they can be ported back. None of our methods requires us to modify the original source code nor the Runtime. Consequently, these solutions do not depend on either Rotor (the Shared Source implementation of the .NET Framework), Mono, or any other open source software.

None of the methods requires the installation of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*.NET Technologies 2006*

Copyright UNION Agency – Science Press,  
Plzen, Czech Republic.

neither the development tools nor the Microsoft .NET Framework SDK on the target machine, and since .NET is a cross-language programming environment, they can be used to generate trace for programs written in any .NET programming language.

The first trace generating method uses the .NET Debugger which we presented in [P6c05] in order to utilize it in our dynamic slicing algorithm, while the second approach exploits the capabilities of the .NET

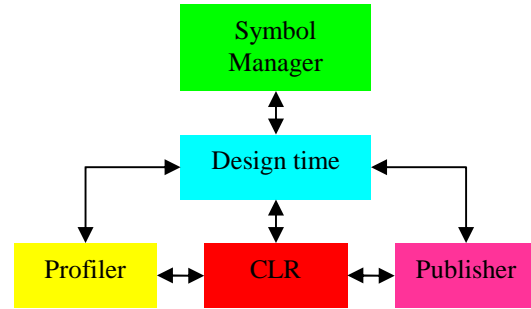
Profiling API and IL code rewriting [Mik03]. It will clear up that only the second method is suitable for large scale multithreaded applications, and the first method is sufficient only for toy programs.

In the next section we will describe the main concepts and the architecture of the *.NET Debugging and Profiling Infrastructure*. In the 3<sup>rd</sup> section we will describe the method that uses the *.NET Debugger* to generate trace, while in the 4<sup>th</sup> section the second solution based on the *.NET Profiler* and *IL code rewriting* technique will be presented. In the 5<sup>th</sup> section we compare these methods and present performance figures with different applications. We will primarily focus on tracing statements of the original source code that appear in the execution path, and will not give detailed description on how to identify variables. However, in the last section we show some ways how the prepared solutions can be complemented to identify variables.

## 2. .NET DEBUGGING AND PROFILING INFRASTRUCTURE

All 20+ .NET languages compile to an intermediate language code called *Common Intermediate Language (CIL)* or simply *Intermediate Language (IL)*. The compiled code is organized into assemblies. Assemblies are portable executables - similar to dll's - with the important difference that assemblies are populated with .NET metadata and IL code instead of normal native code. The .NET metadata holds information about the defined and referenced assemblies, types, methods, class member variables and attributes [ECMA]. IL is a machine-independent, programming language independent, low level, assembly-like language using a stack to transfer data among IL instructions. The IL code is jitted by the .NET CLR (Common Language Runtime) to machine-dependent instructions at runtime.

In the Microsoft world, with the release of .NET, a new Debugging API has also been introduced. Script engines can now compile or interpret code for the Microsoft Common Language Runtime (CLR) instead of integrating debugging capabilities directly into applications through Active Scripting [Pell].



**Figure 1:** CLR Debugging architecture

.NET Debugging Services is not only able to debug every code compiled to IL written in any high level language, but it also provides debugging capabilities for all modern Object Oriented languages.

The .NET CLR supports two types of debugging modes: out-of-process and in-process.

Out-of-process debuggers run in a separate process providing common debugger functionality.

In-process debuggers are used for inspecting the run-time state of an application and for collecting profiling information. These kinds of debuggers (profilers) do not have the ability to control the process or handle events like stepping, breakpoints, etc.

The CLR Debugging Services are implemented as a set of some 70+ COM interfaces, which include the *design-time application*, the *symbol manager*, the *publisher* and the *profiler*.

The *design-time interface* is responsible for handling debugging events. It is implemented separated from the CLR while the host application must reside in a different process. The application has a separate thread for receiving debugger events that run in the context of the debugged application. When a debug event occurs (assembly loaded, thread started, breakpoint reached, etc.) the application halts and the debugger thread notifies the debugging service through callback functions.

The *symbol manager* is responsible for interpreting the program database (PDB) files that contain data used to describe code for the modules being executed. The debugger also uses assembly metadata that also holds useful information described some paragraphs before. The PDB files contain debugging information and are generated only when the compiler is explicitly forced to do so. Besides enabling the unique identification of program elements like classes, methods, variables and statements, the metadata and the program database can also be used to retrieve their original position in the source code.

The *publisher* is responsible for enumerating all running managed processes in the system.

The *profiler* tracks application performance and resources used by running managed processes. The profiler runs in-process of the inspected application and can be used to handle events like module and class loading/unloading, jitting, method calls, events related to exceptions and garbage collection performance.

### 3. .NET DEBUGGER WAY TO INSTRUMENT APPLICATIONS

To employ the *Debugger* first we set a breakpoint to the entry point of our application and we step along each executing statement until the end. The step (or step-in) debugging operation goes along sequence points in the original source code. Sequence points which can be identified using metadata and the program database divide the statements in high-level programming languages.

The CLR Debugger API called *ICorDebug* [Stall] is implemented by native COM interfaces. It can be directly reached from managed or unmanaged code but there are also higher level managed wrapper classes used by MDbg [Stall], the managed debugger part of the Microsoft .NET Framework 2.0 SDK with full source code.

Using these interfaces we can start a process for debugging and register our managed or unmanaged callback functions. As mentioned earlier, querying run-time information of program variables is another important application.

The structure of our solution:

1. Low level managed COM Wrapper
2. High level managed API of the previous
3. Application employing the previous to generate runtime execution trace

The 1<sup>st</sup> and the 2<sup>nd</sup> layer of our solution is not implemented by us rather we borrowed it from MDbg that is freely usable and provided by Microsoft.

The low level managed COM Wrapper (1<sup>st</sup> layer) represents a COM marshaling code that is used to call native Debugging API functions and is written in IL. It resides in the corapi2 folder in MDbg's source tree.

The high level managed API (2<sup>nd</sup> layer) provides an easy-to-use higher level managed wrapper to the underlying layer and it is written in C# 2.0. Sometimes it uses properties instead of methods, and dispatches native debugging events as managed events. It resides in the corapi folder of MDbg's source tree.

Our application based on these APIs is downloadable from <http://avalon.inf.elte.hu/src/netdebug/>.

In the implementation first we create the process to be run but do not start it. A Debugger event is raised at every module load. When the module containing the user entry point (Main method) is loaded we set a breakpoint at this entry point. After loading the process and setting the breakpoint we let the application to run. At this point the process is really created and the *OnCreateProcess* event is raised by the Debugger. In the handler of this event we set the state of the application being debugged to running and start a while loop which is allowed to run while the application is alive. When the breakpoint previously set is encountered the *OnBreakPoint* debug event is raised. In the handler of this debug event an *AutoResetEvent* called *eventComplete* is set and we wait for *eventModState* to be set. The handler of *OnStepComplete* Debugger event does exactly the same.

Afterwards the while loop is doing the following three things:

1. Waits for the *eventComplete* event which is set by the Debugger event handlers
2. *doStepIn* operation is called as described later
3. Sets the *eventModState* event

Between setting the *eventComplete* event and waiting for the *eventModState* event the *doStepIn* method runs which requires/sets the following information at every step:

1. The IL instruction pointer
2. The current function token and module
3. Which sequence point belongs to the current IL instruction
4. The target of the next step

The IL instruction pointer, the function token and the module can be easily queried from the *CorFrame* object which can be queried from the current thread. The sequence points are required to output the actual source line and source column to the trace and to define the next step using the *StepRange* method of *CorStepper*. The sequence points and the target of the next step are static properties therefore we cache them so that they can be queried by the *GetSequencePoints* and *GetRanges* method of the current *ISymbolMethod* interface accordingly. At the first and last sequence point of each function we log a function enter and leave event in the trace.

Unfortunately, this approach is not able to correctly handle multithreaded application because we are not able to step from one thread to another and the debugger does not notify us about thread switches.

#### 4. .NET PROFILER WAY TO INSTRUMENT APPLICATIONS

Basically, this approach explores all sequence points in all methods of all classes and all modules of the application being profiled and inserts trace method calls defined in an outer assembly at every sequence point at IL code level [Mik03].

The .NET Profiler provides a COM interface called *ICorProfilerCallback2* exposing a set of callbacks which can be implemented as a COM class. The implementer is not allowed to use any managed programming language, otherwise the Profiler would profile itself. Consequently we have chosen the C++ language to demonstrate this approach.

We have used some other COM interfaces also like *ISymUnmanagedReader*, *ISymUnmanagedMethod*, *IMetaDataImport* and *ICorProfilerInfo2* while the standard classes implementing these interfaces were instantiated using Microsoft's ATL (Active Template Library).

From the 70+ Profiler events provided by the *ICorProfilerCallback2* interface we have used only two: *ModuleLoadFinished* and *ClassLoadFinished*.

### 4.1. Tracing Methods: Implementation and Referencing

In this section we will discuss what tracing methods we are using, how they log and the way we are referencing them.

```
public static void DoFunc(uint startLine,
    uint startColumn, uint endLine, uint endColumn,
    uint functionID, uint action)
{
    try
    {
        lock (lockObj)
        {
            char act = 'E';
            if (action == 2)
                act = 'L';
            sw.WriteLine("{6}T{5}{4}{0}:{1}-{2}:{3}",
                startLine, startColumn, endLine,
                endColumn, act, functionID,
                Thread.CurrentThread.ManagedThreadId);
        }
    }
    catch { }
```

**Listing 1:** Trace method

We created a module (assembly) called *TracerModule* and placed a static class called *Tracer* in it containing only static methods.

Listing 1 illustrates the trace method executed at every method entry (first sequence point executed) and leave (last sequence point, which is always executed unless exception has been thrown).

The first four parameters represent the position of the

sequence point in the source code, the fifth parameter represents the unique function identifier and the action code (1 for E(nte)r, 2 for L(eave)). Because the tracer is prepared for multithreaded applications, we *lock* on a static object and output the unique managed thread identifier at every step. At intra-function sequence points the trace method gets only the first four parameters and does not output any function identifier and action code.

If we are intended to call a method placed in an outer module we have to reference the assembly containing that method, the class and the method itself. We decided not to modify the original program in any way so we have to add these references to the in-memory metadata of every assembly at runtime. The best place to do this is the *ModuleLoadFinished* Profiler event.

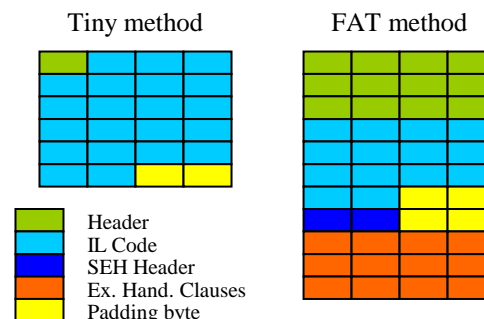
Through the *DefineAssemblyRef* method of the *IMetaDataAssemblyEmit* interface, and the *DefineTypeRefByName* and the *DefineMemberRef* methods of *IMetaDataEmit2* interface we are able to add these references to the in-memory metadata of assemblies and receive their *token* values. When adding these references they are specified simply by their names, the function token is used to call the belonging function at the corresponding sequence points.

## 4.2. Internal Representation of Native .NET Primitives

In this section we will give a general overview of the internal representation of .NET methods, IL instructions and Exception Handling Clauses [Mik03].

#### 4.2.1. Internal representation of .NET methods

Every .NET method has a header, IL code and may have extra padding bytes to maintain DWORD alignment. Optionally, it may have an SEH (Structured Exception Handling) header and Exception Handling Clause.



**Figure 2: Method formats**

A .NET method can be in *Tiny* and in *Fat* format. A Tiny method is smaller than 64 bytes, its stack depth does not exceed 8 slots, contains no local variables,

SEH header and exception handlers. Fat methods overrun one or more of these criterions.

#### 4.2.2. IL instruction types

IL instructions can be divided into several categories based on the number and type of parameters they use:

- have no parameter (dup: duplicates the element on top of the stack; ldc.i4.-1,...ldc.i4.8: load an integer on stack (-1,...8))
- has one integer (8, 16, 32, 64 bits long) parameter (ldc.i4 <int>: load the integer specified by <int> on stack; br <param>, br.s <reloff>: long or short jump to the relative address specified by <reloff>)
- has one token parameter (call <token>: calls the method specified by <token>; box <token>: box a value type with type <token> into an object; ldfld <token>: load the field specified by <token> of the stack-top class on stack)
- multi-parameter instructions (switch <count> <reloff1>...<reloffcount>: based on the stack-top value representing the relative offset parameter index jumps to the chosen relative offset)

#### 4.2.3. Exception Handling Clauses

Every *Fat* method can have one or more exception handlers. Every EHC (Exception Handling Clause) has a header and specifies its *try* and *handler* starting (absolute) offset and length. An EHC can be also in *Tiny* and *Fat* format based on the number of bytes the offset and length properties are used to describe. Obviously each EH offset and length specifies a sequence point beginning and ending position in the IL code-flow.

### 4.3. Let the Game Begin: IL Code Rewriting

Our goal is to change the IL Code of methods before they are jitted to native code. We have chosen the *ClassLoadFinished* Profiler event to perform this operation because in this early stage we are able to enumerate all methods (with the *EnumMethods* method of *IMetaDataImport* interface) of the class just loaded and rewrite the IL code of a whole bunch of methods. The binary data of a method can be retrieved by the *GetILFunctionBody* method of *ICorProfilerInfo2*. After IL code rewriting, necessary space for the new binary data can be allocated using the *Alloc* method of *IMethodMalloc* and the binary data can be set with the *SetILFunctionBody* method of *ICorProfilerInfo2*.

Single-method binary data operations and IL code rewriting can be divided to five steps:

1. Parsing binary data and storing it in custom data

structures

2. Upgrading method and instruction format
3. Insertion of instrumentation code to the IL code-flow
4. Recalculating offsets and lengths
5. Storing new representation in binary format

#### 4.3.1. Parsing binary method data

At first we determine the sequence points of the method being parsed using the *GetSequencePoints* method of *ISymUnmanagedMethod*. This procedure determines the IL- and original source code-level start and end offsets for every sequence point. The first byte of the header describes whether the method is tiny or fat, the function is parsed using this information.

The IL-level offsets of sequence points were determined previously, now the binary data has to be assigned to them and the IL instructions have to be identified based on the binary data at every sequence point. Every category of IL instructions featured in 4.2.2 is able to parse itself and determine its parameters (integer value, token value, multiple parameters). Furthermore it can also generate both a human readable and a binary representation (along with its length) of it.

```
static bool IsFirstLess(int value1, int value2)
{
    if (value1 < value2)
    {
        Console.WriteLine("Yes, first is less");
        return true;
    }
    return false;
}
```

**Listing 2: Simple C# Method**

Consider the simple method in Listing 2. In Table 1 the corresponding sequence points are shown identified by their IL offset, the start and end offsets by line and column numbers.

Index	IL offset	Start offset	End offset
0	0	25,1	25,2
1	1	26,3	26,23
2	9	0xfeefee,0	0xfeefee,0
3	12	27,3	27,4
4	13	28,7	28,47
5	24	29,7	29,19
6	28	31,3	31,16
7	32	32,1	32,2

**Table 1: Sequence Point Offsets**

Sequence point at index 2 petted FeeFee does not have a real source code level offset just helps us to jump out if the predicate fails.

The IL code in Listing 4 illustrates the internal representation of method in Listing 2. The numbering on the left indicates the IL offsets while the numbers



right to the branch instructions (*brtrue.s*, *br.s*) represents absolute target offset, relative target offset, target sequence point and target instruction index at the target sequence point. Parameters of *ldstr* and *call* instructions are of type string and functions tokens respectively. The absolute target offset of branch instructions identified by target IL instruction has to be calculated from the instruction offset and the relative target offset.

If exist, the EHCs are also parsed [Mik03].

```
0: nop
1: ldarg 0
2: ldarg 1
3: clt
5: ldc.i4 0
6: ceq
8: stloc 1
9: ldloc 1
10: brtrue.s 28 (16) [tsp: 6, til: 0]
12: nop
13: ldstr 1879048193
18: call 167772181
23: nop
24: ldc.i4 1
25: stloc 0
26: br.s 32 (4) [tsp: 7, til: 0]
28: ldc.i4 0
29: stloc 0
30: br.s 32 (0) [tsp: 7, til: 0]
32: ldloc 0
33: ret
```

**Listing 4:** Human Readable Output of Internal Method Representation

#### 4.3.2. Upgrading method and instruction format

In case of *Tiny* method format the header is upgraded to represent a *Fat* format because we can easily overrun the limitations of *Tiny* format.

The short branch instructions (*brtrue.s*, *br.s*, *bge.un.s*, etc.) are converted to their long pairs (*brtrue*, *br*, *bge.un*, etc.) because we cannot guarantee that the relative branch lengths will remain within the numeric representation barriers after inserting some instrumentation instructions between the branch instructions and their targets.

Tiny Exception Handling Clauses are also upgraded to store offset and length values in DWORD format because the limitation of original WORD (offset) and BYTE (length) can be easily overrun after instrumentation code insertion.

#### 4.3.3. Instrumentation code insertion

Now we have the *Token* IDs of Trace methods, queried the IL and source code level offsets and lengths of sequence points and converted the binary data to upgraded IL instruction flow. Now we examine how the methods called *DoFunc* (in Listing 1) and its pair called *DoTrace* can be parameterized and called. While *DoFunc* is intended to use at method enter and leave, *DoTrace* handles intra-function sequence points.

As we have mentioned earlier, IL instructions are able to parse themselves therefore we create a BYTE

array to store binary data which can be easily parsed and stored in the same type of container where the original instructions are stored.

The parameters of the method to be called are loaded on the stack using the *ldc.i4* instruction (opcode 0x20) in order of parameters and the *Token ID* of method is given as the parameter of *call* instruction (opcode 0x28). The possible instruction (*ldc.i4.1*, or *ldc.i4.2*) at index 25 surely having a one byte opcode (0x17 or 0x18) loads 1 for enter or 2 for leave on stack respectively.

```
BYTE insertFuncInst[31];
insertFuncInst[0] = 0x20; //ldc.i4, start line
insertFuncInst[5] = 0x20; //ldc.i4, start column
insertFuncInst[10] = 0x20; //ldc.i4, end line
insertFuncInst[15] = 0x20; //ldc.i4, end column
insertFuncInst[20] = 0x20; // ldc.i4, func. id
insertFuncInst[25] = 0x0; // ldc.i4.1 or ldc.i4.2
insertFuncInst[26] = 0x28; // call
*((DWORD *) (insertFuncInst+27)) =
    tracerDoFuncMethodTokenID;
```

**Listing 3:** Binary representation of trace method call

The above parameters are dynamically substituted depending on the data of the current sequence point and a unique function ID (generated by an own counter) while the function token can be preset since it is module (and not function) dependent.

In the intra-function sequence points only the data of sequence points is substituted and the thread ID is queried at each step, the function ID and other information are irrelevant here. The substituted binary data is parsed and converted to IL instructions and inserted into the beginning of the IL code container of every sequence point.

#### 4.3.4. Recalculating offsets and lengths

Since the IL instruction flow is altered by inserting extra instructions the target offsets of branch instructions and the start offset and length properties of Exception Handling Clauses have to be recalculated.

A target offset of a branch instruction can point to the first instruction of a sequence point and can point to other than the first instruction. If the original branch target offset pointed to the first instruction of a sequence point then we change the target offset to the newly created first instruction in order to run instrumentation after jumps also. If the original branch target pointed to other then the first instruction then we leave it to target to the same instruction as before.

Any IL instruction in our representation can calculate its length, so we can easily recalculate the new offsets of IL instructions and sequence points for the branch targets also.

The offset and length properties of Exception Handling Clauses can be calculated similarly.

```

0: ldc.i4 25      |112: ldc.i4 47
5: ldc.i4 1       |117: call 167772194
10: ldc.i4 25     |122: ldstr 1879048193
15: ldc.i4 2      |127: call 167772181
20: ldc.i4 3      |132: nop
25: ldc.i4 1      |133: ldc.i4 29
26: call 167772195 |138: ldc.i4 7
31: nop          |143: ldc.i4 29
32: ldc.i4 26     |148: ldc.i4 19
37: ldc.i4 3      |153: call 167772194
42: ldc.i4 26     |158: ldc.i4 1
47: ldc.i4 23     |159: stloc 0
52: call 167772194 |160: br 197 (32)
57: ldarg 0       |165: ldc.i4 31
58: ldarg 1       |170: ldc.i4 3
59: clt           |175: ldc.i4 31
61: ldc.i4 0      |180: ldc.i4 16
62: ceq           |185: call 167772194
64: stloc 1       |190: ldc.i4 0
65: ldloc 1       |191: stloc 0
66: brtrue 165 (94) |192: br 197 (0)
71: ldc.i4 27     |197: ldc.i4 32
76: ldc.i4 3      |202: ldc.i4 1
81: ldc.i4 27     |207: ldc.i4 32
86: ldc.i4 4      |212: ldc.i4 2
91: call 167772194 |217: ldc.i4 3
96: nop           |222: ldc.i4 2
97: ldc.i4 28     |223: call 167772195
102: ldc.i4 7     |228: ldloc 0
107: ldc.i4 28    |229: ret

```

**Listing 5:** Altered IL code of IsFirstLess method

#### 4.3.5. Storing the instrumented method

Now we have the instrumented method represented in our data structures. The job is to store the data and IL code back in binary format following the specification. The binary data can be restored to the CLR by using the method described in 4.3.

## 5. COMPARISON OF METHODS AND TEST RESULT

In the previous sections we have presented two different methods for generating runtime execution trace of .NET-based applications.

None of the methods require us to modify the applications being tested. Both methods can be accomplished to produce trace information about the value of accessed variables of any type, and identify reference variables. With the help of the Debugger, reference variables can be identified by their Object Id, but obtaining this Id requires many time consuming operations [Stall]. Using the Profiler's IL code rewriting capabilities it is also possible to identify reference variables, and much faster than with the Debugger. A value type variable is always identifiable by the sequence point occurrence it was created in.

The Debugger is unable to notify us about thread switches and the step-in operation is unable to jump through threads therefore it is not possible to handle multithreaded applications. To the contrary, using the Profiler we are able to log the thread's ID at every sequence point of the application.

In order to make the Debugger work we have to attach it to the process we intend to instrument. To

use the Profiler, it is required to register it as a COM component using the *regsvr32* command and set two environment variables in the process, user or system context to enable the Profiler in that context. Set *Cor\_Enable\_Profiling* to 0x1 and *Cor\_Profiler* to the GUID or ProgID of our object implementing the *ICorProfilerCallback2* interface.

We demonstrate the performance of the methods through four applications. The first two use only few class library calls so they are intended to measure the pure performance. The third application uses much more but very short, while the last one uses many and long class library calls.

The character of the four applications:

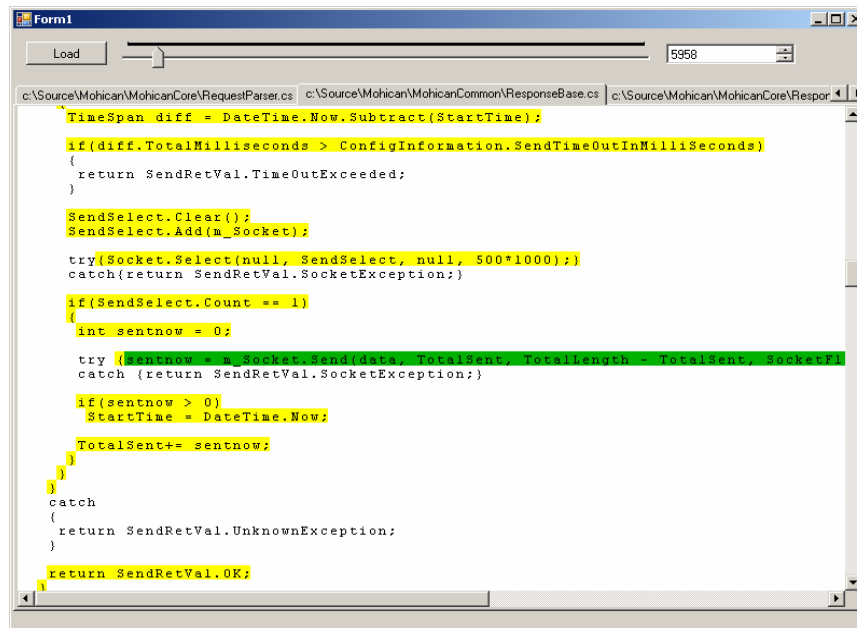
1. Counter is a simple application that calculates the sum of numbers from 1 to 10000 and prints a dot at each step on the screen by implementing the addition in a separate function and uses only few class library calls, but a lot of integer operations which are implemented by native IL instructions.
2. ITextSharp is an open source PDF library. In our test we created a basic PDF document. It uses very few class library calls and a lot of string operations which are implemented by native IL instructions.
3. DiskReporter recursively walks the directory tree from a previously specified path and creates an XML report. In our test 3141 directories and 12257 files were enumerated. It uses more, but short library calls (xml node and attribute operations, file property query).
4. Mohican is a small HTTP server using multiple threads for serving requests. In our test Mohican served a 1.3MB HTML document referencing 20 different pictures. It uses many and long class library calls (mainly network and file access).

App. name	Normal run	Debugger trace	Profiler trace	No. of SPs
Counter	00:00.17	01:53:92	00:01.34	110,034
ITextSharp	00:01:02	98:11.32	02:33:50	2,825,242
Disk-Reporter	00:05.46	24:04.42	00:11.76	316,196
Mohican	00:01.37	n/a	00:01.89	175,434

**Table 2:** Test results

Table 2 shows the performance comparison of the normal application run, the run under the control of the Debugger and the Profiler in mm:ss.ii format. The last column contains the number of source code statements executed.

It can be seen that applications containing few class library calls perform poor under the control of both the Debugger and the Profiler, while applications containing many class library calls perform better.



**Figure 3:** Visualizing the trace

Applications containing long class library calls (like any real world enterprise application) perform well under the control of the Profiler. Unfortunately the Debugger could not be tested (because of multithreading).

The runtime trace generated by the Profiler can be visualized using a Winform application as shown in Figure 3 (the trace of Mohican). The code fragment in green (darker) shows the statement executed at an arbitrary step of the application. Statements in yellow (lighter) have already been executed, while white statements have not yet been traversed.

## 6. CONCLUSION AND FURTHER WORK

In this paper we have shown how to utilize the .NET Debugging and Profiling Infrastructure to generate runtime execution trace of large applications and analyzed both method using programs of different characteristic. We can conclude that although the method based on the Debugger is easier to implement, the Profiler is much more suitable for tracing large scale, multithreaded applications.

Therefore, we plan to advance on the Profiler way. The first and most important thing is to extend our framework to identify variables in the order as local variables, method arguments and class variables appear. We can insert instrumentation code after any variable load and before any variable store operation. The on-stack-top variables can be duplicated by the *dup* IL instruction in order to consume them in the parameter of a trace method call.

There are some language elements and CLR features

which we currently do not support like exceptions, nested classes, anonymous methods, generic types and methods, application domains.

## 7. REFERENCES

- [Agr91a] H. Agrawal and J. R. Horgan. Dynamic program slicing. In SIGPLAN Notices No. 6, pages 246-256, 1990.
- [Bes01a] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, T. Gyimóthy. Dynamic slicing method for maintenance of large C programs, CSMR 2001, pages 105-113.
- [ECMA] ECMA C# and Common Language Infrastructure Standards  
<http://msdn.microsoft.com/netframework/ecma/>
- [Mar03a] K. Maruyama, M. Terada, Timestamp Based Execution Control for C and Java Programs, AADEBUG, 2003
- [Mik03] A. Mikunov, Rewrite MSIL Code on the Fly with the .NET Framework Profiling API, MSDN magazine, issue September 2003,  
<http://msdn.microsoft.com/msdnmag/issues/03/09/NETProfilingAPI/>
- [Póc05] K. Pócza, M. Biczó, Z. Porkoláb. Cross-language Program Slicing in the .NET Framework, Journal of .NET Technologies, 2005
- [Stall] Mike Stall's .NET Debugging Blog, <http://blogs.msdn.com/jmstall/>, 2004-2006
- [Tip95a] F. Tip, A survey of program slicing techniques. Journal of Programming Languages, 3(3):121-189, Sept. 1995.
- [Zha03a] X. Zhang, R. Gupta, Y. Zhang. Precise dynamic slicing algorithms. Proc. International Conference on Software Engineering, pages 319-329, 2003