Towards Detailed Trace Generation Using the Profiler in the .NET Framework*

Krisztián Pócza, Mihály Biczó, and Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages
Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary.
kpocza@kpocza.net, mihaly.biczo@t-online.hu, gsd@elte.hu

Abstract. Effective runtime trace generation is vital for understanding, analyzing, and maintaining large-scale applications. In this paper an effective detailed runtime trace generation approach is introduced for the .NET platform. The non-intrusive method is based on the .NET Profiler; consequently, neither additional development tools, nor the .NET Framework SDK is required to be installed on the target system. The method is applied to a test set of real-size executables and compared by performance and applicability to the original program.

1 Introduction

Runtime trace generation of applications and analyzing runtime traces is a worthwhile task to investigate the cause of arising malfunctions and accidental crashes.

In order to produce reliable applications, for a program developer it is important to investigate programs using a debugger, so that erroneous parts of the program, instructions and variables getting incorrect values can be detected. However, there are many situations where a simple debugger fails to fulfill this task or we are not allowed to use a debugger [6]. Furthermore, multithreaded applications or applications producing incorrect behavior only under heavy load often may not be debugged correctly on development machines.

The most common research area where detailed runtime traces can be used is dynamic program slicing [1, 3, 8, 10, 11]. Besides being widely studied in the academic field, dynamic program slicing has industrial applications as well. 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, the programmers can identify bugs more precisely and at a much earlier stage.

In this article we show a method for generating source code statement level runtime traces for applications hosted by the Microsoft .NET Framework 2.0. The method does not require the modification of the original source code nor the .NET Runtime. Consequently, these solutions do not depend on either Rotor [12] (the Shared Source implementation of the .NET Framework) or Mono [13]

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

(open source, multiplatform implementation of the .NET Framework and the C# compiler under GPL sponsored by Novell).

The method does not require the installation of any development tools. A further benefit of the proposed solution is language independence: since .NET is a cross-language programming environment, it can be used to generate traces for programs written in any .NET-compliant programming language like C#, Visual Basic, etc.

The trace generating method exploits the capabilities of the .NET Profiling API and intermediate language (IL) code rewriting. The basics of this approach were presented in [9]. This paper complements our solution with the ability of tracing variable reads and writes, and gives a clearer overview of the solution.

The structure of the paper is the following: in the next section we describe the main concepts and the architecture of the .NET Debugging and Profiling Infrastructure. In Section 3 the basic functionalities of the solution like utilizing the Profiler API, IL code rewriting and sequence point level trace generation will be presented. In Section 4 variable level tracing capabilities and our implementation will be shown. In Section 5 we analyze the performance of the method and present performance figures with different applications. In the last section we examine how the prepared solution can be extended to identify reference and output parameters, pointers, etc.

2 The .NET Debugging and Profiling Infrastructure

There are over 40 .NET languages, all of whom can be compiled 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 [5]. 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 just-in-time compiled by the .NET CLR (Common Language Runtime) to machine-dependent instructions at runtime.

.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, while in-process debuggers can be applied for inspecting the run-time state of an application and for collecting profiling information.

The CLR Debugging Services are implemented as a set of more than 70 COM (Component Object Model) interfaces, which include the design-time interface, the symbol manager, the publisher and the profiler.

Figure 1 shows the architecture of the CLR Debugging Infrastructure, different modules and the connections between them.

The design-time interface is responsible for handling debugging events. It is implemented separated from the CLR, while the host application must reside in

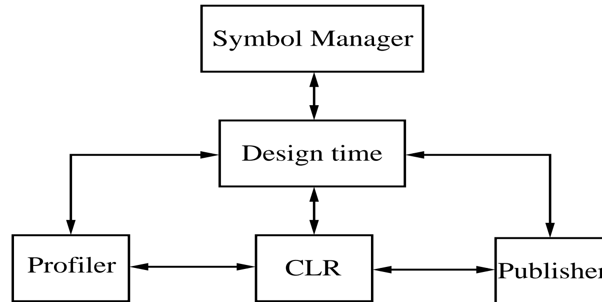

Fig. 1. CLR Debugging Infrastructure

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 the assembly metadata that also holds useful information described earlier. 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" (just-in-time compilation), method calls, and events related to exceptions and garbage collection performance.

3 Sequence Point Level Runtime Trace Generation

In this section we will recall our method of sequence point level runtime trace generation that was previously described in [9]. A sequence point is a point in time during the execution of a program. At each sequence point it is guaranteed that all of the previous operations have been performed and their side effects are visible to operations to be executed afterwards, however, no operation after the sequence point has been called when execution reaches the sequence point. In other words side effects of operations performed prior to the sequence point are guaranteed to be visible to operations performed after it. Often ; and , characters sign sequence points. We will shortly introduce the internal IL and

metadata representation of methods compiled into .NET assemblies, and show how we can insert new IL code sequences to create detailed runtime trace.

Basically, this approach employs the .NET Profiler and 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 [7].

The .NET Profiler provides a COM interface called `ICorProfilerCallback2` exposing a set of callbacks (events) which can be implemented.

We have used some other COM interfaces to dig into assemblies. From the 70+ Profiler events provided by the `ICorProfilerCallback2` interface only two had to be employed: `ModuleLoadFinished` and `ClassLoadFinished`.

3.1 Tracing Methods: Implementation and Referencing

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

We created an outer assembly called `TracerModule` and added a static class called `Tracer` containing only static methods.

```
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: Tracing Method

The above source code illustrates the trace method executed at every method entry (first sequence point executed) and leave (last sequence point, which is always executed unless an exception has been thrown).

The `WriteLine` method writes out the trace lines. The first parameter of WriteLine contains a formatting expression where `{q}` references to the *qth* parameter indexed from zero after the formatting expression. The first four parameters represent the position of the sequence point in the source code, the

`functionID` parameter represents the unique function identifier, and the `act` parameter gives the action code (1 for E(nter), 2 for L(eave)). Since the tracer is prepared for multithreaded applications, we lock on a static object and output the unique managed thread identifier at every step using the last parameter. At intra-function sequence points the trace method gets only the first four parameters and the thread identifier, and does not output any function identifier or action code.

If we intend 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.

3.2 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 [7].

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 a SEH (Structured Exception Handling) header and Exception Handling Clause.

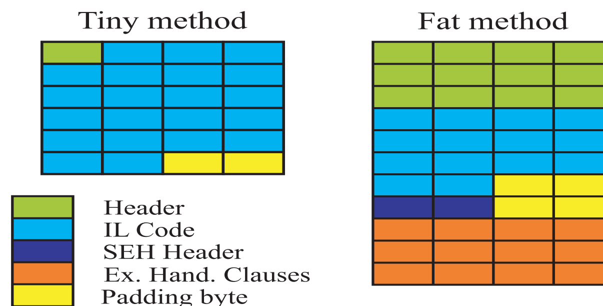

Fig. 2. Internal Representation of .NET Methods

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.

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

– instructions with no parameter (e.g. dup: duplicates the element on top of the stack; ldc.i4.-1, ... , ldc.i4.8: load an integer on stack (-1, 8)).
– instructions with an integer parameter (8, 16, 32, 64 bits long) parameter (e.g. 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>)
– instructions with a token parameter (e.g. 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 the stack)
– multi-parameter instructions (e.g. switch <count> <reloff1> ... <reloffcount>: based on the stack-top value representing the relative offset parameter index jumps to the chosen relative offset).

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 EHC offset and length specifies a sequence point beginning and ending position in the IL code-flow.

3.3 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 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 a single call. After IL code rewriting, necessary space for the new binary data has to be allocated and the newly generated binary data have to be loaded.

Single-method binary data operations and IL code rewriting can be divided into 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.

Parsing Binary Method Data Firstly, we determine the IL- and original source code-level start and end offsets of every sequence point of the method being parsed. The first byte of the header describes whether the method is in Tiny or Fat format, 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.

```
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 sequence points of the source code in Listing 2 are 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; it is needed by the Framework.

The IL code in the Listing 3 illustrates the internal representation of method in Listing 2. The numbering on the left indicates the IL offsets while numbers to the right of the branch instructions (`brtrue.s, br.s`) represents absolute target offset, relative target offset, target sequence point(tsp) and target instruction index(til) in 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 EHCs exist, they are also parsed [7].

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`,

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

Listing 3: IL Code of Method in Listing 1

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.

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 IL instruction flow where branch instructions are converted to their long pairs. We examine how the instrumentation methods can be parameterized and called. While DoFunc (Listing 1) is intended to be used at method enter and leave, another method is needed which handles intra-function sequence points. First we create a BYTE array to store binary data of IL instructions intended to do instrumentation method parameterization and call, which can be easily integrated into our current representation format 4.

The parameters of the method to be called are loaded on the stack using the ldc.i4 instruction (opcode 0x20) in the order of parameters, and the Token ID of the method is given as the parameter of the 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 the stack, respectively.

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). 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.

```
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 4: Code Inserting Instrumentation IL Instructions

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. The 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 than 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.

Listing 5 shows the altered IL instruction sequence that performs runtime trace generation presented in Listing 3.

The original IL code of the method shown in Listing 3 started with a nop instruction. Now this instruction is preceded with some constant loading and method call instructions. The constants store the source code level starting and ending column and line numbers of the entry point of the method and indicate method enter explained in 3.3.3. From index 32 to 52, from index 71 to 91, etc. intra-function sequence point tracing instructions can be seen. The instructions from index 197 to 223 will trace the fact that the method is being left.

Storing the Instrumented Method Now we have the instrumented method represented using our data structures. The challenge here is to convert the data and IL code back to binary format following the specification. The binary data can be restored to the CLR by using the method described in Section 3.3.

```
0:  ldc.i4 25          59: clt               112: ldc.i4 47
5:  ldc.i4 1           61: ldc.i4 0          117: call 167772194
10: ldc.i4 25          62: ceq               122: ldstr879048193
15: ldc.i4 2           64: stloc 1           127: call 167772181
20: ldc.i4 3           65: ldloc 1           132: nop
25: ldc.i4 1           66: brtrue 165(94)    133: ldc.i4 29
26: call 167772195     71: ldc.i4 27         138: ldc.i4 7
31: nop                76: ldc.i4 3          143: ldc.i4 29
32: ldc.i4 26          81: ldc.i4 27         148: ldc.i4 19
37: ldc.i4 3           86: ldc.i4 4          153: call 167772194
42: ldc.i4 26          91: call 167772194    158: ldc.i4 1
47: ldc.i4 23          96: nop               159: stloc 0
52: call 167772194     97: ldc.i4 28         160: br 197 (32)
57: ldarg 0            102: ldc.i4 7         165: ldc.i4 31
58: ldarg 1            107: ldc.i4 28        170: ldc.i4 3
```

Listing 5: IL Code with Instrumentation Calls Added

4 Variable Level Runtime Trace Generation

Our aim is to develop a method that generates trace for every variable usage and definition and can be integrated into our current framework. Consider the five steps of IL code rewriting introduced in Section 3.3. The trace generation for variables has to be done between the 3rd and 4th step just before recalculating branch instructions relative targets.

In this section we categorize variables and variable types using different aspects, show some enumeration methods that enumerate and put the variables into these categories and show the setup and structure of the instrumentation code generating the trace.

4.1 Variable Categories

Variables can be distinguished based on the following properties:

1. Place of the definition of a variable
2. Value or Reference variable type
3. System class of a variable type

A variable can be defined as a *local* variable, can be a *parameter* of a method and can be a *class member* field.

The .NET CLR specification divides the variable types into two categories: *value* and *reference* types. Value types are stored on the stack, cannot have null value, the Garbage Collector (GC) does not take care of them and the assignment operator creates a copy of the variable. Reference types are stored on the heap, can have null values, memory deallocation is done by the GC and the assignment operator does not copy the variable content but only the memory

reference. (There is another subcategory called *nullable* which are value types that can have null values.)

Through the system class of a variable type we mean how deep a variable is integrated in the CLR, i.e. what is its representation method in the metadata. The elementary types like int, string, boolean, object, etc. are represented in the metadata as a simple byte, while complex types, like classes and structure are represented by their metadata token.

4.2 Enumerating Variables and Variable Types

When calling a trace generation method we need to pass the variable being inspected as a parameter to the trace generation method which accepts the parameter as an object type which is the base of all types in .NET. In case of reference type variables the parameter passing is easy because the reference type variables are automatically cast to an object. In case of value type variables an explicit boxing is needed which requires us to call the box instruction with the variable type token parameter. For consistency the type token of reference types are also enumerated.

The enumeration of variables and their variable types is a two phase process. The first phase is performed when an assembly module is loaded. In this phase the defined and referenced types and type members (class and structure fields) are enumerated. The second phase is performed when a method to be instrumented is encountered and local and parameter variables and their variable types are enumerated.

As mentioned earlier the element types are stored in the metadata as a simple byte, but they are also referenced from the *mscorlib* system assembly and they also have an assembly-dependent type token which can be extracted from the metadata.

The COM interfaces for metadata manipulation do not provide any API for low level type information inspection (class members, local variable definitions and parameter definitions). What can be obtained using a built-in API is binary data called *signature blob* which has to be parsed using a custom low-level parser.

Visiting David Broman's CLR Profiling API Blog we can find an implementation of a signature blob parser which could be integrated into our framework with some modifications [4].

Now we have all resources to enumerate the variables and variable types. At module load time (in the `ModuleLoadFinished` profiler event) the element types are enumerated and their metadata tokens are identified. The next step is that all referenced and defined complex types and their members are enumerated. Now the signature blobs of members and metadata tokens are identified (using the signature blob parser).

4.3 Inserting Variable Usage Instrumentation Code

We have 6 different instrumentation method instances which are also static members of the same instrumentation class introduced in Section 3.1. The methods

are implemented for the following 6 purposes: local variable usage, local variable definition, parameter variable usage, parameter variable definition, field variable usage, field variable definition. We do not have to create more methods based on other category types, because these methods can accept the variable as an object type parameter. This parameter can be used to query all necessary type information including whether the variable is a value type or a reference type, moreover elementary and also complex types can be passed. The value of the variable can also be queried.

To determine the exact place where an instrumentation method call has to be inserted, we have to inspect the IL code and find all instructions which load a variable (variable usage) on the stack or stores the topmost item on the stack to a variable (variable definition). These instructions in our current implementation are: `ldloc` and `ldloc.s` for local variable usage, `stloc` and `stloc.s` for local variable definition, `ldarg` and `ldarg.s` for parameter variable usage, starg and starg.s for parameter variable definition, ldfld for field variable usage, and stfld for field variable definition. The instrumentation method call has to be inserted after every variable usage instruction and before every variable definition instruction.

Because the instrumentation method call consumes an instance of the variable just loaded or intended to store on stack, the topmost stack element has to be duplicated using the dup IL instruction. Value types have to be boxed and some additional information about the variable has to be also passed to the instrumentation method like parameter and local variable index or field variable metadata token.

The following code (Listing 6) illustrates the instructions performing the setup of a instrumentation method call specialized for local value type variable usage trace generation:

```
BYTE insertTraceLocalUseValueInst[16];
insertTraceLocalUseValueInst[0] = 0x25; //dup
insertTraceLocalUseValueInst[1] = 0x8c; //box
insertTraceLocalUseValueInst[6] = 0x20; //ldc.i4
insertTraceLocalUseValueInst[11] = 0x28; // call
*((DWORD *)(insertTraceLocalUseValueInst+12)) =
                        tracerDoLocalVarUseMethodTokenID;
```

Listing 6: Local Variable Usage Trace Insert

The above parameters are dynamically substituted. At byte index 2 the `box` instruction gets the metadata token of the value type variable and at byte index 7 the `ldc.i4` instruction gets the local variable index. The setup code for local reference type variable usage is almost the same but there is no need of the box instruction. The other setup codes are almost the same; therefore they are not presented in detail in this paper.

The instrumentation method contains the code presented in Listing 7:

```
public static void DoLocalVarUse(object var, uint index)
{
  try
  {
    lock (lockObj)
    {
      if(var != null)
      {
        Type t = var.GetType();
        if (t.IsValueType)
           sw.WriteLine("{3}LUV{0}:{1}:{2}", index, t,
             var.ToString(), Thread.CurrentThread.ManagedThreadId);
        else
          sw.WriteLine("{4}LU{2}R{0}:{1}:{3}", index, t,
                      System.Runtime.CompilerServices.
                      RuntimeHelpers.GetHashCode(var),
                      var.ToString(),
                      Thread.CurrentThread.ManagedThreadId);
      }
      else
      {
         sw.WriteLine("{1}LU{0}NULL", index,
                      Thread.CurrentThread.ManagedThreadId);
      }
    }
  }
  catch { }
}
```

Listing 7: Local Variable Use Trace Method

This method is prepared for multithreaded operations, because there is a lock statement and a unique thread identifier is sent to the output in every step. We distinguish not null and null variables. If the variable is not null, then it is decided whether we deal with a value or a reference type variable. In case of value types we log some identification information (like local variable index), the variable type, the output of the `ToString` method and the managed thread identifier while in case of reference types we log the hash code in addition to the same properties as logged for value types. The `GetHashCode` function of many .NET types are overridden so we query the hash code of the variable as it would be an object using the `System.Runtime.CompilerServices.RuntimeHelpers` class. If the variable is null then we only log some identification information and the managed thread identifier.

5 The Results

We demonstrate the performance of our method that generates sequence point and variable level runtime trace 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 class library calls, while the last one uses many and long class library calls.

The characters of the four applications are as follows:

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 3245 directories and 12849 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).

Application name	Normal run	Profiler trace	No. of trace items
Counter	00:00.17	00:22.11	1 700 014
ITextSharp	00:01.02	43:24.97	5 364 020
DiskReporter	00:02.56	00:14.60	850 345
Mohican	00:00.52	00:04.63	97 353

Table 2. Test results

Table 2 shows the performance comparison of the normal application run and the Profiler in mm:ss.ii format. The last column contains the number of lines in the generated trace.

It can be seen that applications containing few class library calls perform poor under the control of the Profiler (like ITextSharp which employs many short string operations), while applications containing many class library calls (DiskReporter) perform better. Applications containing long class library calls (Mohican in the measurement and any real world enterprise application) perform well under the control of the Profiler.

A fragment of the runtime trace generated by our Framework while running Mohican can be seen in Listing 8.

```
 1: 3FU11429296R67108918:System.String:
 2: 3PU9040679R1:System.String:HTTP/1.1 200 OK
 3: 3FD19473824R67108918:System.String:HTTP/1.1 200 OK
 4: 3T133:4-133:48
 5: 3FU19473824R67108918:System.String:HTTP/1.1 200 OK
 6: 3LDV1:System.Boolean:True
 7: 3T136:4-136:16
 8: 3LDV0:System.Boolean:True
 9: 3T69L137:3-137:4
10: 3LUV0:System.Boolean:True
11: 3T45:5-45:63
12: 3T78E425:3-425:4
13: 3T426:4-426:30
14: 3LD62619566R0:System.String:text/plain
15: 3T429:4-429:5
16: 3T430:5-430:41
17: 3PU61646925R1:System.String:C:\Source\Mohican\wwwroot/index.html
18: 3LDV1:System.Int32:37
19: 3T432:5-432:50
20: 3PU61646925R1:System.String:C:\Source\Mohican\wwwroot/index.html
21: 3LUV1:System.Int32:37
```

Listing 8: Generated Trace

To make the snippet from the generated runtime trace clearer we explain the first, the second and the fourth lines. The first number always indicates the thread number on the instruction was executed (in out case thread number *3*). *FU* indicates class member field usage. Number *11429296* is the hash code of the variable. *R* indicates that it is a reference type variable while number *67108918* is the IL level token value of the variable. The type of the variable is string. In the second line *PU* indicates a parameter usage that has hash code value *9040679* moreover the trace shows that we are using the first parameter that is reference type string variable. The value of the variable is *HTTP/1.1 200 OK*. Line 4 indicates that the execution reaced a new expression that resides on line number *133* from column number *4* to column number *48*.

6 Conclusion and Further Work

In this paper we have shown how to utilize the .NET Profiler to generate runtime execution trace of large applications. We can conclude that the Profiler is suitable for tracing real-world, multithreaded applications. Therefore, we plan to advance on this tracing method. The first and most important thing to do with variables is to extend our framework to better identify method parameters passed as output and reference, identify memory allocations and array item accesses.

There are some language elements and CLR features which we currently do not support like exceptions, nested classes, C# anonymous methods, generic types, generic methods and application domains.

In some cases the current implementation suffers from performance issues, therefore it is important to optimize the trace generation mechanism [2].

References

1. Hiralal Agrawal, and Joseph R. Horgan. *Dynamic Program Slicing.* Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation. pages 246-256, White Plains, NY, June 1990.
2. Matthew Arnold and Barbara G. Ryder. *A Framework for Reducing the Cost of Instrumented code.* In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, pages 168-179, ACM Press, 2001.
3. 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.
4. David Broman's CLR Profiling API Blog, Info about the Common Language Runtime's Profiling API, `http://blogs.msdn.com/davbr/archive/2005/10/13/480864.aspx`
5. ECMA C# and Common Language Infrastructure Standards, `http://msdn.microsoft.com/netframework/ecma/`
6. K. Maruyama, M. Terada. *Timestamp Based Execution Control for C and Java Programs (Automated Debugging).* AADEBUG, September 2003.
7. 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/`
8. K. Pócza, M. Biczó, Z. Porkoláb. *Cross-language Program Slicing in the .NET Framework.* Proceedings of the 3rd .NET Technologies Conference, pages 141-150, 2005.
9. K. Pócza, M. Biczó, Z. Porkoláb. *Towards Effective Runtime Trace Generation Techniques in the .NET Framework.* Short Communication Papers Proceedings of the 4th .NET Technologies Conference, pages 9-16, 2006.
10. F. Tip. *A survey of program slicing techniques.* Journal of Programming Languages, 3(3):121-189, Sept. 1995.
11. X. Zhang, R. Gupta, Y. Zhang. *Precise Dynamic Slicing Algorithms.* Proceedings of International Conference on Software Engineering, pages 319-329, 2003.
12. Shared Source Common Language Infrastructure 2.0 Release: `http://msdn.microsoft.com/net/sscli/`
13. Mono: project, `http://www.mono-project.com/Main_Page`