Time Profilers

Although performance counters and ETW events offer a great amount of insight into the performance of Windows applications, there's often a lot to be gained from more intrusive tools—profilers—that inspect application execution time at the method and line level (improving upon ETW stack trace collection support). In this section we introduce some commercial tools and see the benefits they bring to the table, bearing in mind that more powerful and accurate tools sustain a bigger measurement overhead.

Throughout our journey into the profiler world we will encounter numerous commercial tools; most of them have several readily available equivalents. We do not endorse any specific tool vendor; the products demonstrated in this chapter are simply the profilers we use most often and keep in our toolbox for performance investigations. As always with software tools, your mileage may vary.

The first profiler we consider is part of Visual Studio, and has been offered by Microsoft since Visual Studio 2005 (Team Suite edition). In this chapter, we'll use the Visual Studio 2012 profiler, which is available in the Premium and Ultimate editions of Visual Studio.

Visual Studio Sampling Profiler

The Visual Studio sampling profiler operates similarly to the `PROFILE` kernel flag we've seen in the ETW section. It periodically interrupts the application and records the call stack information on every processor where the application's threads are currently running. Unlike the kernel ETW provider, this sampling profiler can interrupt processes based on several criteria, some of which are listed in Table 2-2.

Table 2-2. *Visual Studio Sampling Profiler Events (Partial List)*

Capturing samples using the Visual Studio profiler is quite cheap, and if the sample event interval is wide enough (the default is 10,000,000 clock cycles), the overhead on the application's execution can be less than 5%. Moreover, sampling is very flexible and enables attaching to a running process, collecting sample events for a while, and then disconnecting from the process to analyze the data. Because of these traits, sampling is the recommended approach to begin a performance investigation for CPU bottlenecks—methods that take a significant amount of CPU time.

When a sampling session completes, the profiler makes available summary tables in which each method is associated with two numbers: the number of *exclusive samples*, which are samples taken while the method was currently executing on the CPU, and the number of *inclusive samples*, which are samples taken while the method was currently executing or anywhere else on the call stack. Methods with many exclusive samples are the ones responsible for the application's CPU utilization; methods with many inclusive samples are not directly using the CPU, but call out to other methods that do. (For example, in single-threaded applications it would make sense for the `Main` method to have 100% of the inclusive samples.)

RUNNING THE SAMPLING PROFILER FROM VISUAL STUDIO

The easiest way to run the sampling profiler is from Visual Studio itself, although (as we will see later) it also supports production profiling from a simplified command line environment. We recommend that you use one of your own applications for this experiment.

1. In Visual Studio, click the Analyze ➤ Launch Performance Wizard menu item.
2. On the first wizard page, make sure the "CPU sampling" radio button is selected and click the Next button. (Later in this chapter we'll discuss the other profiling modes; you can then repeat this experiment.)
3. If the project to profile is loaded in the current solution, click the "One or more available projects" radio button and select the project from the list. Otherwise, click the "An executable (.EXE file)" radio button. Click the Next button.
4. If you selected "An executable (.EXE file)" on the previous screen, direct the profiler to your executable and provide any command line arguments if necessary, then click the Next button. (If you don't have your own application handy, feel free to use the JackCompiler.exe sample application from this chapter's source code folder.)
5. Keep the checkbox "Launch profiling after the wizard finishes" checked and click the Finish button.
6. If you are not running Visual Studio as an administrator, you will be prompted to upgrade the profiler's credentials.
7. When your application finishes executing, a profiling report will open. Use the "Current View" combo box on the top to navigate between the different views, showing the samples collected in your application's code.

For more experimentation, after the profiling session completes make sure to check out the Performance Explorer tool window (Analyze ➤ Windows ➤ Performance Explorer). You can configure sampling parameters (e.g. choosing a different sample event or interval), change the target binary, and compare multiple profiler reports from different runs.

Figure 2-12 shows a summary window of the profiler's results, with the most expensive call path and the functions in which most exclusive samples have been collected. Figure 2-13 shows the detail report, in which there are a few methods responsible for most CPU utilization (have a large number of exclusive samples). Double-clicking a method in the list brings up a detailed window, which shows the application's source color-coded with the lines in which most samples were collected (see Figure 2-14).

Figure 2-12*. Profiler report, Summary view—the call path responsible for most of the samples and the functions with most exclusive samples*

Figure 2-13*. Functions view, showing the functions with the most exclusive samples. The System.String.Concat function is responsible for twice as many samples as any other function*

Figure 2-14*. Function Details view, showing the functions calling and called by the JackCompiler.CompilationOutputTextWriter.WriteLine function. In the function's code, lines are highlighted according to the percent of inclusive samples they accumulated*

Caution It may appear that sampling is an accurate technique for measuring CPU utilization. You might hear claims that go as far as "if this method had 65% of the exclusive samples, then it ran for 65% of the time". Because of the statistical nature of sampling, such reasoning is treacherous and should be avoided in practical use. There are several factors that can contribute to the inaccuracy of sampling results: CPU clock rates can change hundreds of times every second during application execution, such that the correlation between the number of samples and actual CPU time is skewed; a method can be "missed" (underrepresented) if it happens not to be running when many samples were taken; a method can be overrepresented if it happens to be running when many samples were taken but finished quickly every time. To summarize, you should not consider the results of a sampling profiler to be an exact representation of where CPU time was spent, but rather a general outline of the application's main CPU bottlenecks.

The Visual Studio profiler offers more information in addition to the exclusive/inclusive sample tables for every method. We recommend that you explore the profiler's windows yourself—the Call Tree view shows the hierarchy of calls in the application's methods (compare to PerfMonitor's top down analysis, Figure 2-8), the Lines view displays sampling information on the line level, and the Modules view groups methods by assembly, which can lead to quick conclusions about the general direction in which to look for a performance bottleneck.

Because all sampling intervals require the application thread that triggers them to be actively executing on the CPU, there is no way to obtain samples from application threads that are blocked while waiting for I/O or synchronization mechanisms. For CPU-bound applications, sampling is ideal; for I/O-bound applications, we'll have to consider other approaches that rely on more intrusive profiling mechanisms.

Visual Studio Instrumentation Profiler

The Visual Studio profiler offers another mode of operation, called *instrumentation profiling*, which is tailored to measuring overall execution time and not just CPU time. This makes it suitable for profiling I/O-bound applications or applications that engage heavily in synchronization operations. In the instrumentation profiling mode, the profiler modifies the target binary and embeds within it measurement code that reports back to the profiler accurate timing and call count information for every instrumented method.

For example, consider the following method:

```
public static int InstrumentedMethod(int param) {
  List< int > evens = new List < int > ();
  for (int i = 0; i < param; ++i) {
   if (i % 2 == 0) {
   evens.Add(i);
   }
  }
  return evens.Count;
}
```

During instrumentation, the Visual Studio profiler modifies this method. Remember that instrumentation occurs at the binary level—your source code is *not* modified, but you can always inspect the instrumented binary with an IL disassembler, such as .NET Reflector. (In the interests of brevity, we slightly modified the resulting code to fit.)

```
public static int mmid = (int)
  Microsoft.VisualStudio.Instrumentation.g_fldMMID_2D71B909-C28E-4fd9-A0E7-ED05264B707A;
public static int InstrumentedMethod(int param) {
  _CAP_Enter_Function_Managed(mmid, 0x600000b, 0);
  _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa000018);
  _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
  List < int > evens = new List < int > ();
  for (int i = 0; i < param; i++) {
   if (i % 2 == 0) {
   _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa000019);
   evens.Add(i);
   _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
   }
  }
  _CAP_StartProfiling_Managed(mmid, 0x600000b, 0xa00001a);
  _CAP_StopProfiling_Managed(mmid, 0x600000b, 0);
  int count = evens.Count;
  _CAP_Exit_Function_Managed(mmid, 0x600000b, 0);
  return count;
}
```

The method calls beginning with `_CAP` are interop calls to the VSPerf110.dll module, which is referenced by the instrumented assembly. They are the ones responsible for measuring time and recording method call counts. Because instrumentation captures every method call made out of the instrumented code and captures method enter and exit locations, the information available at the end of an instrumentation run can be very accurate.

When the same application we've seen in Figures 2-12, 2-13, and 2-14 is run under instrumentation mode (you can follow along—it's the JackCompiler.exe application), the profiler generates a report with a Summary view that contains similar information—the most expensive call path through the application, and the functions with most individual work. However, this time the information is not based on sample counts (which measure only execution on the CPU); it is based on precise timing information recorded by the instrumentation code. Figure 2-15 shows the Functions view, in which inclusive and exclusive times measured in milliseconds are available, along with the number of times the function has been called.

Figure 2-15. The Functions view: System.String.Concat no longer appears to be the performance bottleneck, as our attention shifts to JackCompiler.Tokenizer.NextChar and JackCompiler.Token..ctor. The first method was called almost a million times

Tip The sample application used to generate Figures 2-12 and 2-15 is not entirely CPU-bound; in fact, most of its time is spent blocking for I/O operations to complete. This explains the difference between sampling results, which point towards `System.String.Concat` as the *CPU* hog, and the instrumentation results, which point towards `JackCompiler.Tokenizer.NextChar` as the *overall* performance bottleneck.

Although instrumentation seems like the more accurate method, in practice you should try to keep to sampling if most of your application's code is CPU-bound. Instrumentation limits flexibility because you must instrument the application's code prior to launching it, and cannot attach the profiler to a process that was already running. Moreover, instrumentation has a non-negligible overhead—it increases code size significantly and places a runtime overhead as probes are collected whenever the program enters or exits a method. (Some instrumentation profilers offer a line-instrumentation mode, where each line is surrounded by instrumentation probes; these are even slower!)

As always, the biggest risk is placing too much trust in the results of an instrumentation profiler. It is reasonable to assume that the number of calls to a particular method does not change because the application is running with instrumentation, but the time information collected may still be significantly skewed because of the profiler's overhead, despite any attempts by the profiler to offset the instrumentation costs from the final results. When used carefully, sampling and instrumentation can offer great insight into where the application is spending time, especially when you compare multiple reports and take note whether your optimizations are yielding fruit.

Advanced Uses of Time Profilers

Time profilers have additional tricks up their sleeves that we haven't examined in the previous sections. This chapter is too short to discuss them in considerable detail, but they are worth pointing out to make sure you don't miss them in the comforts of Visual Studio's wizards.

Sampling Tips

As we saw in the Visual Studio Sampling Profiler section, the sampling profiler can collect samples from several types of events, including cache misses and page faults. In Chapters 5 and 6 we will see several examples of applications that can benefit greatly from improving their memory access characteristics, primarily around minimizing cache misses. The profiler will prove valuable in analyzing the number of cache misses and page faults exhibited by these applications and their precise locations in code. (When using instrumentation profiling, you can still collect CPU counters such as cache misses, instructions retired, and mispredicted branches. To do so, open the performance session properties from the Performance Explorer pane, and navigate to the CPU Counters tab. The collected information will be available as additional columns in the report's Functions view.)

The sampling profiling mode is generally more flexible than instrumentation. For example, you can use the Performance Explorer pane to attach the profiler (in sampling mode) to a process that is already running.

Collecting Additional Data While Profiling

In all profiling modes, you can use the Performance Explorer pane to pause and resume the data collection when the profiler is active, and to generate markers that will be visible in the final profiler report to discern more easily various parts of the application's execution. These markers will be visible in the report's Marks view.

Tip The Visual Studio profiler even has an API that applications can use to pause and resume profiling from code. This can be used to avoid collecting data from uninteresting parts of the application, and to decrease the size of the profiler's data files. For more information about the profiler APIs, consult the MSDN documentation at http://msdn.microsoft.com/en-us/library/bb514149(v=vs.110).aspx.

The profiler can also collect Windows performance counters and ETW events (discussed earlier in this chapter) during a normal profiling run. To enable these, open the performance session properties from Performance Explorer, and navigate to the Windows Events and Windows Counters tabs. ETW trace data can only be viewed from the command line, by using the `VSPerfReport /summary:ETW` command line switch, whereas performance counter data will appear in the report's Marks view in Visual Studio.

Finally, if Visual Studio takes a long time analyzing a report with lots of additional data, you can make sure it was a one-time performance hit: after analysis completes, right-click the report in Performance Explorer and choose "Save Analyzed Report". Serialized report files have the .vsps file extension and open instantaneously in Visual Studio.

Profiler Guidance

When opening a report in Visual Studio, you might notice a section called Profiler Guidance which contains numerous useful tips that detect common performance problems discussed elsewhere in the book, including:

- "Consider using `StringBuilder` for string concatenations"—a useful rule that may help lower the amount of garbage your application creates, thus reducing garbage collection times, discussed in Chapter 4.
- "Many of your objects are being collected in generation 2 garbage collection"—the *mid-life crisis* phenomenon for objects, also discussed in Chapter 4.
- "Override `Equals` and equality operator on value types"—an important optimization for commonly-used value types, discussed in Chapter 3.
- "You may be using Reflection excessively. It is an expensive operation"—discussed in Chapter 10.

Advanced Profiling Customization

Collecting performance information from production environments may prove difficult if you have to install massive tools such as Visual Studio. Fortunately, the Visual Studio profiler can be installed and run in production environments without the entire Visual Studio suite. You can find the profiler setup files on the Visual Studio installation media, in the `Standalone Profiler` directory (there are separate versions for 32- and 64-bit systems). After installing the profiler, follow the instructions at http://msdn.microsoft.com/en-us/library/ms182401(v=vs.110).aspx to launch your application under the profiler or attach to an existing process using the VSPerfCmd.exe tool. When done, the profiler will generate a .vsp file that you can open on another machine with Visual Studio, or use the VSPerfReport.exe tool to generate XML or CSV reports that you can review on the production machine without resorting to Visual Studio.

For instrumentation profiling, many customization options are available from the command line, using the VSInstr.exe tool. Specifically, you can use the START, SUSPEND, INCLUDE, and EXCLUDE options to start and suspend profiling in a specific function, and to include/exclude functions from instrumentation based on a pattern in their name. More information about VSInstr.exe is available on the MSDN at http://msdn.microsoft.com/en-us/library/ms182402.aspx.

Some time profilers offer a remote profiling mode, which allows the main profiler UI to run on one machine and the profiling session to take place on another machine without copying the performance report manually. For example, the JetBrains dotTrace profiler supports this mode of operation through a small remote agent that runs on the remote machine and communicates with the main profiler UI. This is a good alternative to installing the entire profiler suite on the production machines.

Note In Chapter 6 we will leverage the GPU for super-parallel computation, leading to considerable (more than 100×!) speedups. Standard time profilers are useless when the performance problem is in the code that runs on the GPU. There are some tools that can profile and diagnose performance problems in GPU code, including Visual Studio 2012. This subject is outside the scope of this chapter, but if you're using the GPU for graphics or plain computation you should research the tools applicable to your GPU programming framework (such as C++ AMP, CUDA, or OpenCL).

In this section, we have seen in sufficient detail how to analyze the application's execution time (overall or CPU only) with the Visual Studio profiler. Memory management is another important aspect of managed application performance. Through the next two sections, we will discuss allocation profilers and memory profilers, which can pinpoint memory-related performance bottlenecks in your applications.