Built-in Windows Tools

Before we turn to commercial tools that tend to require installation and intrusively measure your application's performance, it's paramount to make sure everything Windows has to offer out-of-the-box has been used to its fullest extent. Performance counters have been a part of Windows for nearly two decades, whereas Event Tracing for Windows is slightly newer and has become truly useful around the Windows Vista time frame (2006). Both are free, present on every edition of Windows, and can be used for performance investigations with minimal overhead.

Performance Counters

Windows performance counters are a built-in Windows mechanism for performance and health investigation. Various components, including the Windows kernel, drivers, databases, and the CLR provide performance counters that users and administrators can consume and understand how well the system is functioning. As an added bonus, performance counters for the vast majority of system components are turned on by default, so you will not be introducing any additional overhead by collecting this information.

Reading performance counter information from a local or remote system is extremely easy. The built-in *Performance Monitor* tool (perfmon.exe) can display every performance counter available on the system, as well as log performance counter data to a file for subsequent investigation and provide automatic alerts when performance counter readings breach a defined threshold. Performance Monitor can monitor remote systems as well, if you have administrator permissions and can connect to them through a local network.

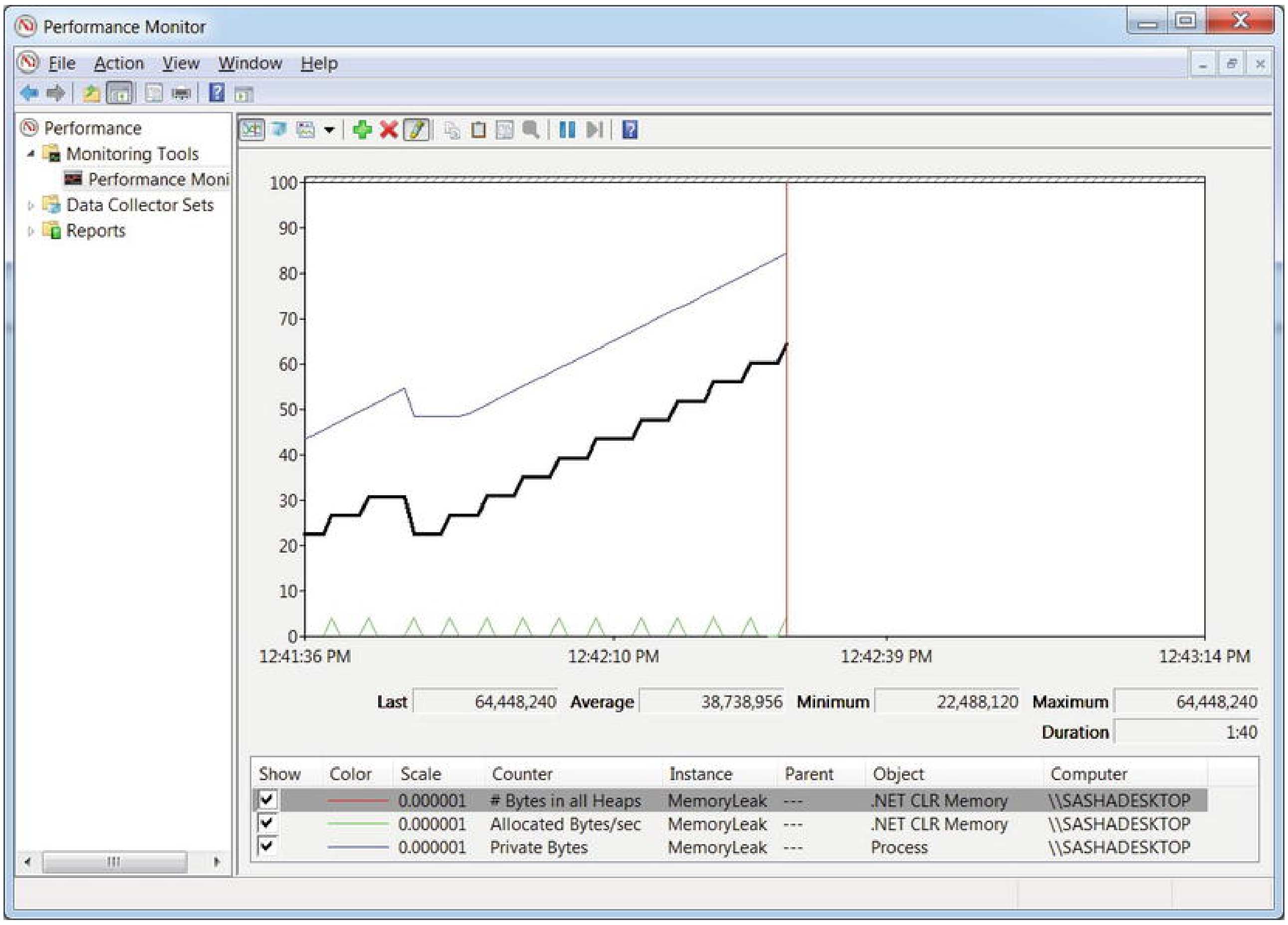
Performance information is organized in the following hierarchy:

- *Performance counter categories* (or *performance objects*) represent a set of individual counters related to a certain system component. Some examples of categories include `.NET CLR Memory`, `Processor Information`, `TCPv4`, and `PhysicalDisk`.
- *Performance counters* are individual numeric data properties in a performance counter category. It is common to specify the performance counter category and performance counter name separated by a slash, e.g., `Process\Private Bytes`. Performance counters have several supported types, including raw numeric information (`Process\Thread Count`), rate of events (`Print Queue\Bytes Printed/sec`), percentages (`PhysicalDisk\% Idle Time`), and averages (`ServiceModelOperation 3.0.0.0\Calls Duration`).
- *Performance counter category instances* are used to distinguish several sets of counters from a specific component of which there are several instances. For example, because there may be multiple processors on a system, there is an instance of the `Processor Information` category for each processor (as well as an aggregated `_Total` instance). Performance counter categories can be multi-instance—and many are—or single-instance (such as the `Memory` category).

If you examine the full list of performance counters provided by a typical Windows system that runs .NET applications, you'll see that many performance problems can be identified without resorting to any other tool. At the very least, performance counters can often provide a general idea of which direction to pursue when investigating a performance problem or inspecting data logs from a production system to understand if it's behaving normally.

Below are some scenarios in which a system administrator or performance investigator can obtain a general idea of where the performance culprit lies before using heavier tools:

- If an application exhibits a memory leak, performance counters can be used to determine whether managed or native memory allocations are responsible for it. The `Process\Private Bytes` counter can be correlated with the `.NET CLR Memory\# Bytes in All Heaps` counter. The former accounts for all private memory allocated by the process (including the GC heap) whereas the latter accounts only for managed memory. (See Figure 2-1.)
- If an ASP.NET application starts exhibiting abnormal behavior, the `ASP.NET Applications` category can provide more insight into what's going on. For example, the `Requests/Sec`, `Requests Timed Out`, `Request Wait Time`, and `Requests Executing` counters can identify extreme load conditions, the `Errors Total/Sec` counter can suggest whether the application is facing an unusual number of exceptions, and the various cache- and output cache-related counters can indicate whether caching is being applied effectively.
- If a WCF service that heavily relies on a database and distributed transactions is failing to handle its current load, the `ServiceModelService` category can pinpoint the problem—the `Calls Outstanding`, `Calls Per Second`, and `Calls Failed Per Second` counters can identify heavy load, the `Transactions Flowed Per Second` counter reports the number of transactions the service is dealing with, and at the same time SQL Server categories such as `MSSQL$INSTANCENAME:Transactions` and `MSSQL$INSTANCENAME:Locks` can point to problems with transactional execution, excessive locking, and even deadlocks.



Last	64,448,240	Average	38,738,956	Minimum	22,488,120	Maximum	64,448,240
						Duration	1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.000001	# Bytes in all Heaps	MemoryLeak	---	.NET CLR Memory	\\SASHADESKTOP
✓	🟩	0.000001	Allocated Bytes/sec	MemoryLeak	---	.NET CLR Memory	\\SASHADESKTOP
✓	🟦	0.000001	Private Bytes	MemoryLeak	---	Process	\\SASHADESKTOP

Figure 2-1. The Performance Monitor main window, showing three counters for a specific process. The top line on the graph is the Process\Private Bytes counter, the middle one is .NET CLR Memory\# Bytes in all Heaps, and the bottom one is .NET CLR Memory\Allocated Bytes/sec. From the graph it is evident that the application exhibits a memory leak in the GC heap

MONITORING MEMORY USAGE WITH PERFORMANCE COUNTERS

In this short experiment, you will monitor the memory usage of a sample application and determine that it exhibits a memory leak using Performance Monitor and the performance counters discussed above.

1. Open Performance Monitor—you can find it in the Start menu by searching for "Performance Monitor" or run perfmon.exe directly.
2. Run the MemoryLeak.exe application from this chapter's source code folder.
3. Click the "Performance Monitor" node in the tree on the left, and then click the green **+** button.
4. From the .NET CLR Memory category, select the # Bytes in all Heaps and Allocated Bytes/sec performance counters, select the MemoryLeak instance from the instance list, and click the "Add >>" button.
5. From the Process category, select the Private Bytes performance counter, select the MemoryLeak instance from the instance list, and click the "Add >>" button.
6. Click the "OK" button to confirm your choices and view the performance graph.
7. You might need to right click the counters in the bottom part of the screen and select "Scale selected counters" to see actual lines on the graph.

You should now see the lines corresponding to the Private Bytes and # Bytes in all Heaps performance counters climb in unison (somewhat similar to Figure 2-1). This points to a memory leak in the managed heap. We will return to this particular memory leak in Chapter 4 and pinpoint its root cause.

Tip There are literally thousands of performance counters on a typical Windows system; no performance investigator is expected to remember them all. This is where the small "Show description" checkbox at the bottom of the "Add Counters" dialog comes in handy—it can tell you that `System\Processor Queue Length` represents the number of ready threads waiting for execution on the system's processors, or that `.NET CLR LocksAndThreads\Contention Rate / sec` is the number of times (per second) that threads attempted to acquire a managed lock unsuccessfully and had to wait for it to become available.

Performance Counter Logs and Alerts

Configuring performance counter logs is fairly easy, and you can even provide an XML template to your system administrators to apply performance counter logs automatically without having to specify individual performance counters. You can open the resulting logs on any machine and play them back as if they represent live data. (There are even some built-in counter sets you can use instead of configuring what data to log manually.)

You can also use Performance Monitor to configure a performance counter alert, which will execute a task when a certain threshold is breached. You can use performance counter alerts to create a rudimentary monitoring infrastructure, which can send an email or message to a system administrator when a performance constraint is violated. For example, you could configure a performance counter alert that would automatically restart your process when it reaches a dangerous amount of memory usage, or when the system as a whole runs out of disk space. We strongly recommend that you experiment with Performance Monitor and familiarize yourself with the various options it has to offer.

CONFIGURING PERFORMANCE COUNTER LOGS

To configure performance counter logs, open Performance Monitor and perform the following steps. (We assume that you are using Performance Monitor on Windows 7 or Windows Server 2008 R2; in prior operating system versions, Performance Monitor had a slightly different user interface—if you are using these versions, consult the documentation for detailed instructions.)

1. In the tree on the left, expand the Data Collector Sets node.
2. Right-click the User Defined node and select New ➤ Data Collector Set from the context menu.
3. Name your data collector set, select the “Create manually (Advanced)” radio button, and click Next.
4. Make sure the “Create data logs” radio button is selected, check the “Performance counter” checkbox, and click Next.
5. Use the Add button to add performance counters (the standard Add Counters dialog will open). When you’re done, configure a sample interval (the default is to sample the counters every 15 seconds) and click Next.
6. Provide a directory which Performance Monitor will use to store your counter logs and then click Next.
7. Select the “Open properties for this data collector set” radio button and click Finish.
8. Use the various tabs to further configure your data collector set—you can define a schedule for it to run automatically, a stop condition (e.g. after collecting more than a certain amount of data), and a task to run when the data collection stops (e.g. to upload the results to a centralized location). When you’re done, click OK.
9. Click the User Defined node, right-click your data collector set in the main pane, and select Start from the context menu.
10. Your counter log is now running and collecting data to the directory you’ve selected. You can stop the data collector set at any time by right-clicking it and selecting Stop from the context menu.

    When you’re done collecting the data and want to inspect it using the Performance Monitor, perform the following steps:
11. Select the User Defined node.
12. Right-click your data collector set and select Latest Report from the context menu.
13. In the resulting window, you can add or delete counters from the list of counters in the log, configure a time range and change the data scale by right-clicking the graph and selecting Properties from the context menu.

Finally, to analyze log data on another machine, you should copy the log directory to that machine, open the Performance Monitor node, and click the second toolbar button from the left (or Ctrl + L). In the resulting dialog you can select the “Log files” checkbox and add log files using the Add button.

Custom Performance Counters

Although Performance Monitor is an extremely useful tool, you can read performance counters from any .NET application using the `System.Diagnostics.PerformanceCounter` class. Even better, you can create your own performance counters and add them to the vast set of data available for performance investigation.

Below are some scenarios in which you should consider exporting performance counter categories:

- You are developing an infrastructure library to be used as part of large systems. Your library can report performance information through performance counters, which is often easier on developers and system administrators than following log files or debugging at the source code level.
- You are developing a server system which accepts custom requests, processes them, and delivers responses (custom Web server, Web service, etc.). You should report performance information for the request processing rate, errors encountered, and similar statistics. (See the ASP.NET performance counter categories for some ideas.)
- You are developing a high-reliability Windows service that runs unattended and communicates with custom hardware. Your service can report the health of the hardware, the rate of your software’s interactions with it, and similar statistics.

The following code is all it takes to export a single-instance performance counter category from your application and to update these counters periodically. It assumes that the `AttendanceSystem` class has information on the number of employees currently signed in, and that you want to expose this information as a performance counter. (You will need the `System.Diagnostics` namespace to compile this code fragment.)

```
public static void CreateCategory() {
  if (PerformanceCounterCategory.Exists("Attendance")) {
   PerformanceCounterCategory.Delete("Attendance");
  }
```

```
    CounterCreationDataCollection counters = new CounterCreationDataCollection();
    CounterCreationData employeesAtWork = new CounterCreationData(
     "# Employees at Work", "The number of employees currently checked in.",
     PerformanceCounterType.NumberOfItems32);
    PerformanceCounterCategory.Create(
     "Attendance", "Attendance information for Litware, Inc.",
     PerformanceCounterCategoryType.SingleInstance, counters);
  }
  public static void StartUpdatingCounters() {
    PerformanceCounter employeesAtWork = new PerformanceCounter(
     "Attendance", "# Employees at Work", readOnly: false);
    updateTimer = new Timer(_ = > {
     employeesAtWork.RawValue = AttendanceSystem.Current.EmployeeCount;
    }, null, TimeSpan.Zero, TimeSpan.FromSeconds(1));
  }
```

As we have seen, it takes very little effort to configure custom performance counters, and they can be of utmost importance when carrying out a performance investigation. Correlating system performance counter data with custom performance counters is often all a performance investigator needs to pinpoint the precise cause of a performance or configuration issue.

Note Performance Monitor can be used to collect other types of information that have nothing to do with performance counters. You can use it to collect configuration data from a system—the values of registry keys, WMI objects properties, and even interesting disk files. You can also use it to capture data from ETW providers (which we discuss next) for subsequent analysis. By using XML templates, system administrators can quickly apply data collector sets to a system and generate a useful report with very few manual configuration steps.

Although performance counters offer a great amount of interesting performance information, they cannot be used as a high-performance logging and monitoring framework. There are no system components that update performance counters more often than a few times a second, and the Windows Performance Monitor won't read performance counters more often than once a second. If your performance investigation requires following thousands of events per second, performance counters are not a good fit. We now turn our attention to *Event Tracing for Windows* (ETW), which was designed for high-performance data collection and richer data types (not just numbers).

Event Tracing for Windows (ETW)

Event Tracing for Windows (ETW) is a high-performance event logging framework built into Windows. As was the case with performance counters, many system components and application frameworks, including the Windows kernel and the CLR, define *providers*, which report *events*—information on the component's inner workings. Unlike performance counters, that are always on, ETW providers can be turned on and off at runtime so that the performance overhead of transferring and collecting them is incurred only when they're needed for a performance investigation.

One of the richest sources of ETW information is the *kernel provider*, which reports events on process and thread creation, DLL loading, memory allocation, network I/O, and stack trace accounting (also known as *sampling*). Table 2-1 shows some of the useful information reported by the kernel and CLR ETW providers. You can use ETW to investigate overall system behavior, such as what processes are consuming CPU time, to analyze disk I/O and network I/O bottlenecks, to obtain garbage collection statistics and memory usage for managed processes, and many other scenarios discussed later in this section.

ETW events are tagged with a precise time and can contain custom information, as well as an optional stack trace of where they occurred. These stack traces can be used to further identify sources of performance and correctness problems. For example, the CLR provider can report events at the start and end of every garbage collection. Combined with precise call stacks, these events can be used to determine which parts of the program are typically causing garbage collection. (For more information about garbage collection and its triggers, see Chapter 4.)

Table 2-1. *Partial List of ETW Events in Windows and the CLR*

Provider	Flag / Keyword	Description	Events (Partial List)
Kernel	`PROC_THREAD`	Creation and destruction of processes and threads	--
Kernel	`LOADER`	Load and unload of images (DLLs, drivers, EXEs)	--
Kernel	`SYSCALL`	System calls	--
Kernel	`DISK_IO`	Disk I/O reads and writes (including head location)	--
Kernel	`HARD_FAULTS`	Page faults that resulted in disk I/O (not satisfied from memory)	--
Kernel	`PROFILE`	Sampling event—a stack trace of all processors collected every 1ms	--
CLR	`GCKeyword`	Garbage collection statistics and information	Collection started, collection ended, finalizers run, ~100KB of memory have been allocated
CLR	`ContentionKeyword`	Threads contend for a managed lock	Contention starts (a thread begins waiting), contention ends
CLR	`JITTracingKeyword`	Just in time compiler (JIT) information	Method inlining succeeded, method inlining failed
CLR	`ExceptionKeyword`	Exceptions that are thrown	--

Accessing this highly detailed information requires an ETW collection tool and an application that can read raw ETW events and perform some basic analysis. At the time of writing, there were two tools capable of both tasks: *Windows Performance Toolkit* (WPT, also known as XPerf), which ships with the Windows SDK, and *PerfMonitor* (do not confuse it with Windows Performance Monitor!), which is an open source project by the CLR team at Microsoft.

Windows Performance Toolkit (WPT)

Windows Performance Toolkit (WPT) is a set of utilities for controlling ETW sessions, capturing ETW events into log files, and processing them for later display. It can generate graphs and overlays of ETW events, summary tables including call stack information and aggregation, and CSV files for automated processing. To download WPT, download the Windows SDK Web installer from http://msdn.microsoft.com/en-us/performance/cc752957.aspx and select only Common Utilities ➤ Windows Performance Toolkit from the installation options screen. After the Windows SDK installer completes, navigate to the `Redist\Windows Performance Toolkit` subdirectory of the SDK installation directory and run the installer file for your system's architecture (Xperf_x86.msi for 32-bit systems, Xperf_x64.msi for 64-bit systems).

Note On 64-bit Windows, stack walking requires changing a registry setting that disables paging-out of kernel code pages (for the Windows kernel itself and any drivers). This may increase the system's working set (RAM utilization) by a few megabytes. To change this setting, navigate to the registry key `HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management`, set the `DisablePagingExecutive` value to the DWORD `0x1`, and restart the system.

The tools you'll use for capturing and analyzing ETW traces are XPerf.exe and XPerfView.exe. Both tools require administrative privileges to run. The XPerf.exe tool has several command line options that control which providers are enabled during the trace, the size of the buffers used, the file name to which the events are flushed, and many additional options. The XPerfView.exe tool analyzes and provides a graphical report of what the trace file contains.

All traces can be augmented with call stacks, which often allow precise zooming-in on the performance issues. However, you don't have to capture events from a specific provider to obtain stack traces of what the system is doing; the `SysProfile` kernel flag group enables collection of stack traces from all processors captured at 1ms intervals. This is a rudimentary way of understanding what a busy system is doing at the method level. (We'll return to this mode in more detail when discussing *sampling* profilers later in this chapter.)

CAPTURING AND ANALYZING KERNEL TRACES WITH XPERF

In this section, you'll capture a kernel trace using XPerf.exe and analyze the results in the XPerfView.exe graphical tool. This experiment is designed to be carried out on Windows Vista system or a later version. (It also requires you to set two system environment variables. To do so, right click Computer, click Properties, click "Advanced system settings" and finally click the "Environment Variables" button at the bottom of the dialog.)

1. Set the system environment variable `_NT_SYMBOL_PATH` to point to the Microsoft public symbol server and a local symbol cache, e.g.: `srv*C:\Temp\Symbols*`http://msdl.microsoft.com/download/symbols

2. Set the system environment variable `_NT_SYMCACHE_PATH` to a local directory on your disk—this should be a *different* directory from the local symbols cache in the previous step.

3. Open an administrator Command Prompt window and navigate to the installation directory where you installed WPT (e.g. `C:\Program Files\Windows Kits\8.0\Windows Performance Toolkit`).

4. Begin a trace with the `Base` kernel provider group, which contains the `PROC_THREAD`, `LOADER`, `DISK_IO`, `HARD_FAULTS`, `PROFILE`, `MEMINFO`, and `MEMINFO_WS` kernel flags (see Table 2-1). To do this, run the following command: `xperf -on Base`

5. Initiate some system activity: run applications, switch between windows, open files—for at least a few seconds. (These are the events that will enter the trace.)

6. Stop the trace and flush the trace to a log file by running the following command: `xperf -d KernelTrace.etl`

7. Launch the graphical performance analyzer by running the following command: `xperfview KernelTrace.etl`

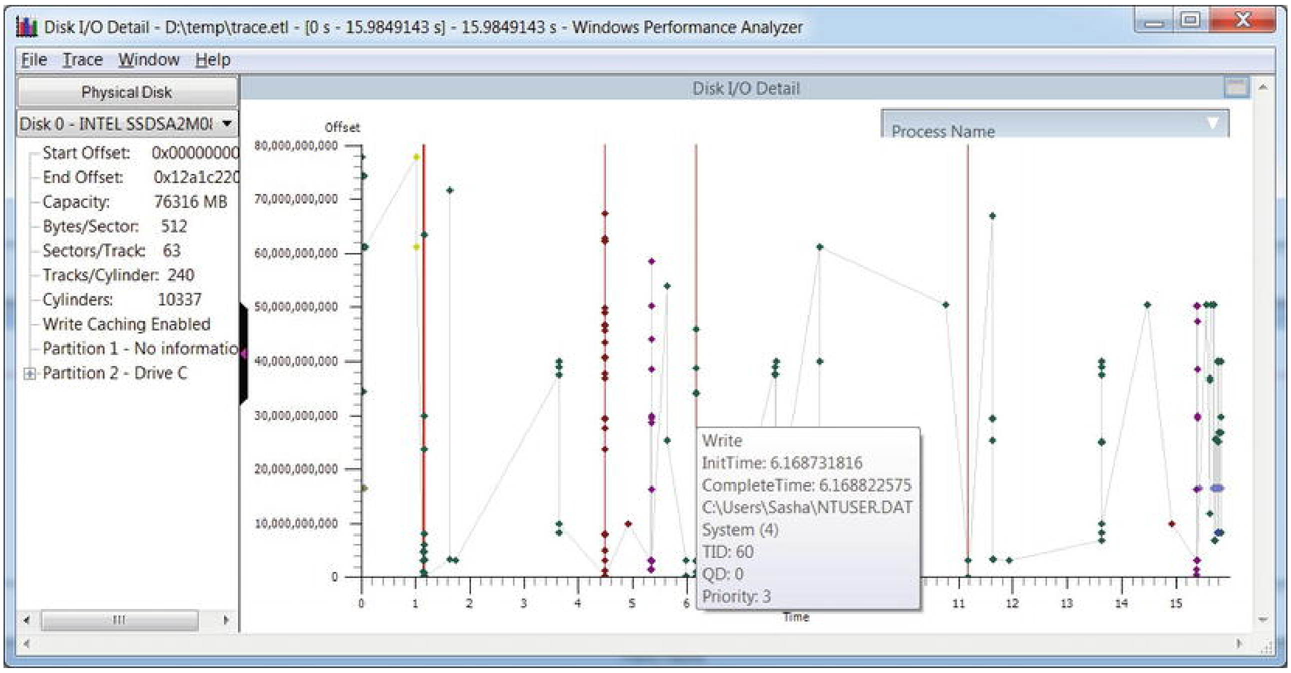
8. The resulting window contains several graphs, one for each ETW keyword that generated events during the trace. You can choose the graphs to display on the left. Typically, the topmost graph displays the processor utilization by processor, and subsequent graphs display the disk I/O operation count, memory usage, and other statistics.

9. Select a section of the processor utilization graph, right click it, and select Load Symbols from the context menu. Right click the selected section again, and select Simple Summary Table. This should open an expandable view in which you can navigate between methods in all processes that had some processor activity during the trace. (Loading the symbols from the Microsoft symbol server for the first time can be time consuming.)

There's much more to WPT than you've seen in this experiment; you should explore other parts of the UI and consider capturing and analyzing trace data from other kernel groups or even your own application's ETW providers. (We'll discuss custom ETW providers later in this chapter.)

There are many useful scenarios in which WPT can provide insight into the system's overall behavior and the performance of individual processes. Below are some screenshots and examples of these scenarios:

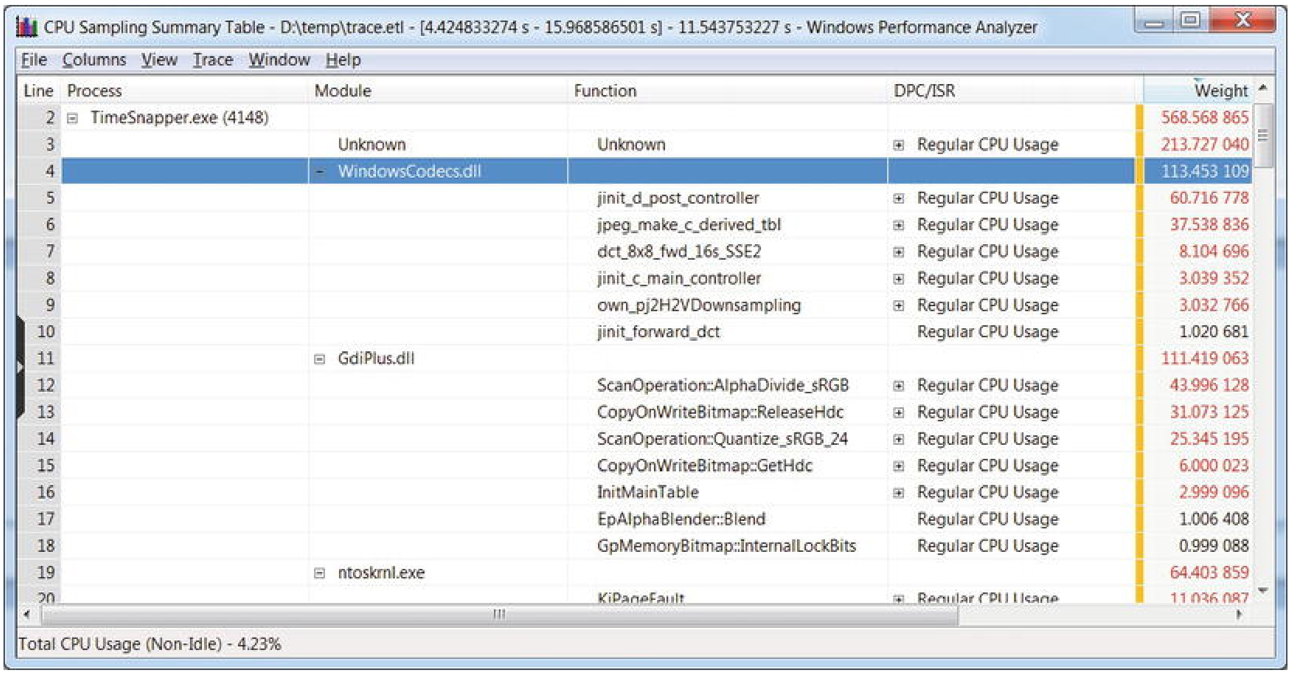
- WPT can capture all disk I/O operations on a system and display them on a map of the physical disk. This provides insight into expensive I/O operations, especially where large seeks are involved on rotating hard drives. (See Figure 2-2.)
- WPT can provide call stacks for all processor activity on the system during the trace. It aggregates call stacks at the process, module, and function level, and allows at-a-glance understanding of where the system (or a specific application) is spending CPU time. Note that managed frames are not supported—we'll address this deficiency later with the PerfMonitor tool. (See Figure 2-3.)
- WPT can display overlay graphs of different activity types to provide correlation between I/O operations, memory utilization, processor activity, and other captured metrics. (See Figure 2-4.)
- WPT can display call stack aggregations in a trace (when the trace is initially configured with the `-stackwalk` command line switch)—this provides complete information on call stacks which created certain events. (See Figure 2-5.)



Disk 0 - INTEL SSDSA2M0(
Start Offset: 0x00000000
End Offset: 0x12a1c220
Capacity: 76316 MB
Bytes/Sector: 512
Sectors/Track: 63
Tracks/Cylinder: 240
Cylinders: 10337
Write Caching Enabled
Partition 1 - No informatio
Partition 2 - Drive C

Write
InitTime: 6.168731816
CompleteTime: 6.168822575
C:\Users\Sasha\NTUSER.DAT
System (4)
TID: 60
QD: 0
Priority: 3

Figure 2-2. Disk I/O operations laid out on a map of the physical disk. Seeks between I/O operations and individual I/O details are provided through tooltips


CPU Sampling Summary Table - D:\temp\trace.etl - [4.424833274 s - 15.968586501 s] - 11.543753227 s - Windows Performance Analyzer

File Columns View Trace Window Help

Line	Process	Module	Function	DPC/ISR	Weight
2	⊟ TimeSnapper.exe (4148)				568.568 865
3		Unknown	Unknown	⊞ Regular CPU Usage	213.727 040
4		− WindowsCodecs.dll			113.453 109
5			jinit_d_post_controller	⊞ Regular CPU Usage	60.716 778
6			jpeg_make_c_derived_tbl	⊞ Regular CPU Usage	37.538 836
7			dct_8x8_fwd_16s_SSE2	⊞ Regular CPU Usage	8.104 696
8			jinit_c_main_controller	⊞ Regular CPU Usage	3.039 352
9			own_pj2H2VDownsampling	⊞ Regular CPU Usage	3.032 766
10			jinit_forward_dct	Regular CPU Usage	1.020 681
11		⊟ GdiPlus.dll			111.419 063
12			ScanOperation::AlphaDivide_sRGB	⊞ Regular CPU Usage	43.996 128
13			CopyOnWriteBitmap::ReleaseHdc	⊞ Regular CPU Usage	31.073 125
14			ScanOperation::Quantize_sRGB_24	⊞ Regular CPU Usage	25.345 195
15			CopyOnWriteBitmap::GetHdc	⊞ Regular CPU Usage	6.000 023
16			InitMainTable	⊞ Regular CPU Usage	2.999 096
17			EpAlphaBlender::Blend	Regular CPU Usage	1.006 408
18			GpMemoryBitmap::InternalLockBits	Regular CPU Usage	0.999 088
19		⊟ ntoskrnl.exe			64.403 859
20			KiPageFault	⊞ Regular CPU Usage	11.036 087

Total CPU Usage (Non-Idle) - 4.23%

Figure 2-3 . Detailed stack frames for a single process (TimeSnapper.exe). The Weight column indicates (roughly) how much CPU time was spent in that frame

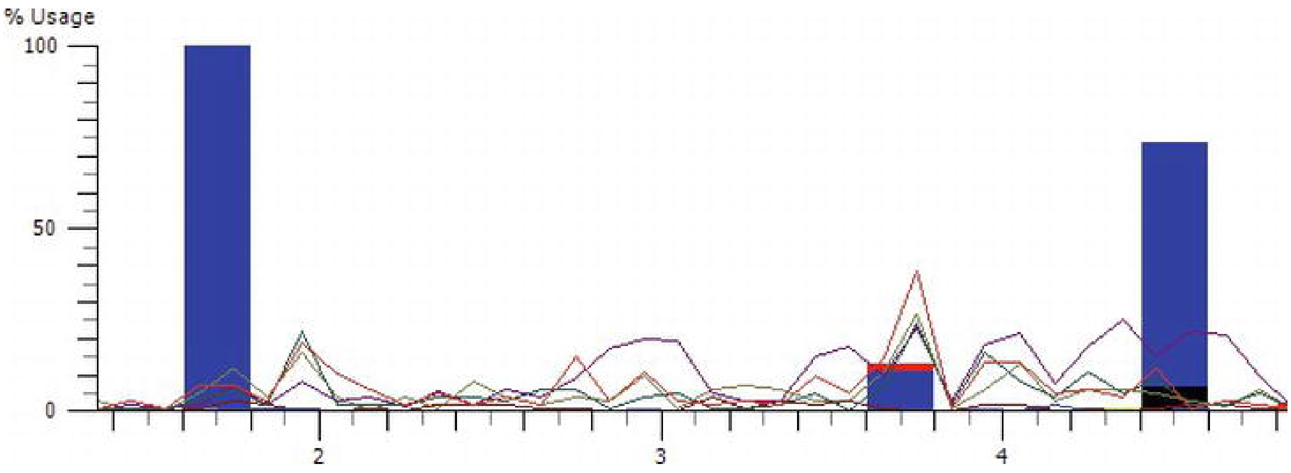
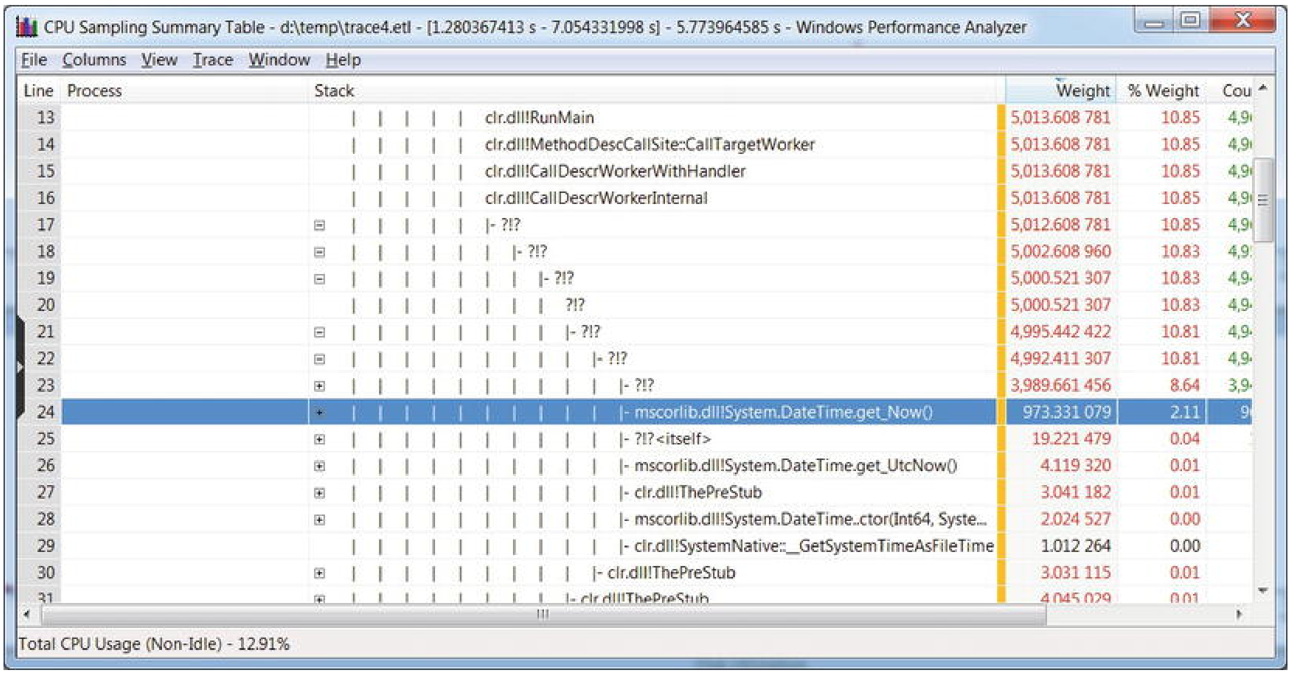


Figure 2-4 . Overlay graph of CPU activity (lines—each line indicates a different processor) and disk I/O operations (columns). No explicit correlation between I/O activity and CPU activity is visible


CPU Sampling Summary Table - d:\temp\trace4.etl - [1.280367413 s - 7.054331998 s] - 5.773964585 s - Windows Performance Analyzer

File Columns View Trace Window Help

Line	Process	Stack	Weight	% Weight	Cou
13		| | | | | clr.dll!RunMain	5,013.608 781	10.85	4,9
14		| | | | | clr.dll!MethodDescCallSite::CallTargetWorker	5,013.608 781	10.85	4,9
15		| | | | | clr.dll!CallDescrWorkerWithHandler	5,013.608 781	10.85	4,9
16		| | | | | clr.dll!CallDescrWorkerInternal	5,013.608 781	10.85	4,9
17		⊟ | | | | | |- ?!?	5,012.608 781	10.85	4,9
18		⊟ | | | | | | |- ?!?	5,002.608 960	10.83	4,9
19		⊟ | | | | | | | |- ?!?	5,000.521 307	10.83	4,9
20		| | | | | | | | ?!?	5,000.521 307	10.83	4,9
21		⊟ | | | | | | | | |- ?!?	4,995.442 422	10.81	4,9
22		⊟ | | | | | | | | | |- ?!?	4,992.411 307	10.81	4,9
23		⊞ | | | | | | | | | | |- ?!?	3,989.661 456	8.64	3,9
24		+ | | | | | | | | | | |- mscorlib.dll!System.DateTime.get_Now()	973.331 079	2.11	9
25		⊞ | | | | | | | | | | |- ?!?<itself>	19.221 479	0.04	
26		⊞ | | | | | | | | | | |- mscorlib.dll!System.DateTime.get_UtcNow()	4.119 320	0.01	
27		⊞ | | | | | | | | | | |- clr.dll!ThePreStub	3.041 182	0.01	
28		⊞ | | | | | | | | | | |- mscorlib.dll!System.DateTime..ctor(Int64, Syste...	2.024 527	0.00	
29		| | | | | | | | | | |- clr.dll!SystemNative::_GetSystemTimeAsFileTime	1.012 264	0.00	
30		⊞ | | | | | | | | | |- clr.dll!ThePreStub	3.031 115	0.01	
31		⊞ | | | | | | | | |- clr.dll!ThePreStub	4.045 029	0.01	

Total CPU Usage (Non-Idle) - 12.91%

Figure 2-5 . Call stack aggregation in the report. Note that managed frames are displayed only partially—the ?!? frames could not be resolved. The mscorlib.dll frames (e.g.

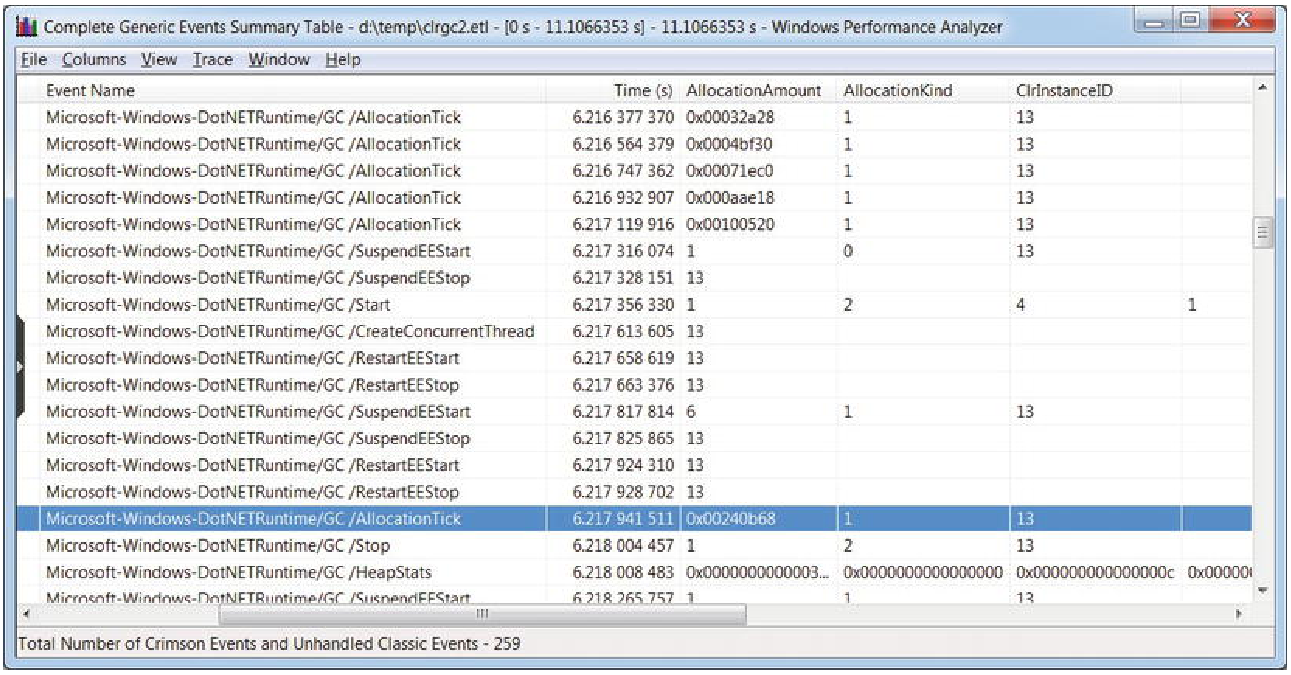
System.DateTime.get_Now()) were resolved successfully because they are pre-compiled using NGen and not compiled by the JIT compiler at runtime

---

Note The latest version of the Windows SDK (version 8.0) ships with a pair of new tools, called Windows Performance Recorder (wpr.exe) and Windows Performance Analyzer (wpa.exe), that were designed to gradually replace the XPerf and XPerfView tools we used earlier. For example, `wpr -start CPU` is roughly equivalent to `xperf -on Diag`, and `wpr -stop reportfile` is roughly equivalent to `xperf -d reportfile`. The WPA analysis UI is slightly different, but provides features similar to XPerfView. For more information on the new tools, consult the MSDN documentation at http://msdn.microsoft.com/en-us/library/hh162962.aspx.

XPerfView is very capable of displaying kernel provider data in attractive graphs and tables, but its support for custom providers is not as powerful. For example, we can capture events from the CLR ETW provider, but XPerfView will not generate pretty graphs for the various events—we'll have to make sense of the raw data in the trace based on the list of keywords and events in the provider's documentation (the full list of the CLR ETW provider's keywords and events is available in the MSDN documentation—http://msdn.microsoft.com/en-us/library/ff357720.aspx).

If we run XPerf with the CLR ETW provider (`e13c0d23-ccbc-4e12-931b-d9cc2eee27e4`) the keyword for GC events (`0x00000001`), and the Verbose log level (`0x5`), it will dutifully capture every event the provider generates. By dumping it to a CSV file or opening it with XPerfView we will be able—slowly—to identify GC-related events in our application. Figure 2-6 shows a sample of the resulting XPerfView report—the time elapsed between the `GC /Start` and `GC /Stop` rows is the time it took a single garbage collection to complete in the monitored application.


Complete Generic Events Summary Table - d:\temp\clrgc2.etl - [0 s - 11.1066353 s] - 11.1066353 s - Windows Performance Analyzer
File Columns View Trace Window Help

Event Name	Time (s)	AllocationAmount	AllocationKind	ClrInstanceID	
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.216 377 370	0x00032a28	1	13	
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.216 564 379	0x0004bf30	1	13	
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.216 747 362	0x00071ec0	1	13	
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.216 932 907	0x000aae18	1	13	
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.217 119 916	0x00100520	1	13	
Microsoft-Windows-DotNETRuntime/GC /SuspendEEStart	6.217 316 074	1	0	13	
Microsoft-Windows-DotNETRuntime/GC /SuspendEEStop	6.217 328 151	13			
Microsoft-Windows-DotNETRuntime/GC /Start	6.217 356 330	1	2	4	1
Microsoft-Windows-DotNETRuntime/GC /CreateConcurrentThread	6.217 613 605	13			
Microsoft-Windows-DotNETRuntime/GC /RestartEEStart	6.217 658 619	13			
Microsoft-Windows-DotNETRuntime/GC /RestartEEStop	6.217 663 376	13			
Microsoft-Windows-DotNETRuntime/GC /SuspendEEStart	6.217 817 814	6	1	13	
Microsoft-Windows-DotNETRuntime/GC /SuspendEEStop	6.217 825 865	13			
Microsoft-Windows-DotNETRuntime/GC /RestartEEStart	6.217 924 310	13			
Microsoft-Windows-DotNETRuntime/GC /RestartEEStop	6.217 928 702	13			
Microsoft-Windows-DotNETRuntime/GC /AllocationTick	6.217 941 511	0x00240b68	1	13	
Microsoft-Windows-DotNETRuntime/GC /Stop	6.218 004 457	1	2	13	
Microsoft-Windows-DotNETRuntime/GC /HeapStats	6.218 008 483	0x0000000000003...	0x0000000000000000	0x000000000000000c	0x00000
Microsoft-Windows-DotNETRuntime/GC /SuspendEEStart	6.218 265 757	1	1	13	

Total Number of Crimson Events and Unhandled Classic Events - 259

Figure 2-6 . Raw report of CLR GC-related events. The selected row displays the GCAllocationTick_V1 event that is raised every time approximately 100KB of memory is allocated

Fortunately, the Base Class Library (BCL) team at Microsoft has identified this deficiency and provided an open source library and tool for analyzing CLR ETW traces, called PerfMonitor. We discuss this tool next.

PerfMonitor

The PerfMonitor.exe open source command line tool has been released by the BCL team at Microsoft through the CodePlex website. At the time of writing the latest release was PerfMonitor 1.5 that can be downloaded from http://bcl.codeplex.com/releases/view/49601. PerfMonitor's primary advantage compared to WPT is that it has intimate knowledge about CLR events and provides more than just raw tabular data. PerfMonitor analyzes GC and JIT activity in the process, and can sample managed stack traces and determine which parts of the application are using CPU time.

For advanced users, PerfMonitor also ships with a library called TraceEvent, which enables programmatic access to CLR ETW traces for automatic inspection. You could use the TraceEvent library in custom system monitoring software to inspect automatically a trace from a production system and decide how to triage it.

Although PerfMonitor can be used to collect kernel events or even events from a custom ETW provider (using the `/KernelEvents` and `/Provider` command line switches), it is typically used to analyze the behavior of a managed application using the built-in CLR providers. Its `runAnalyze` command line option executes an application of your choice, monitors its execution, and upon its termination generates a detailed HTML report and opens it in your default browser. (You should follow the PerfMonitor user guide—at least through the Quick Start section—to generate reports similar to the screenshots in this section. To display the user guide, run `PerfMonitor usersguide`.)

When PerfMonitor is instructed to run an application and generate a report, it produces the following command line output. You can experiment with the tool yourself as you read this section by running it on the JackCompiler.exe sample application from this chapter's source code folder.

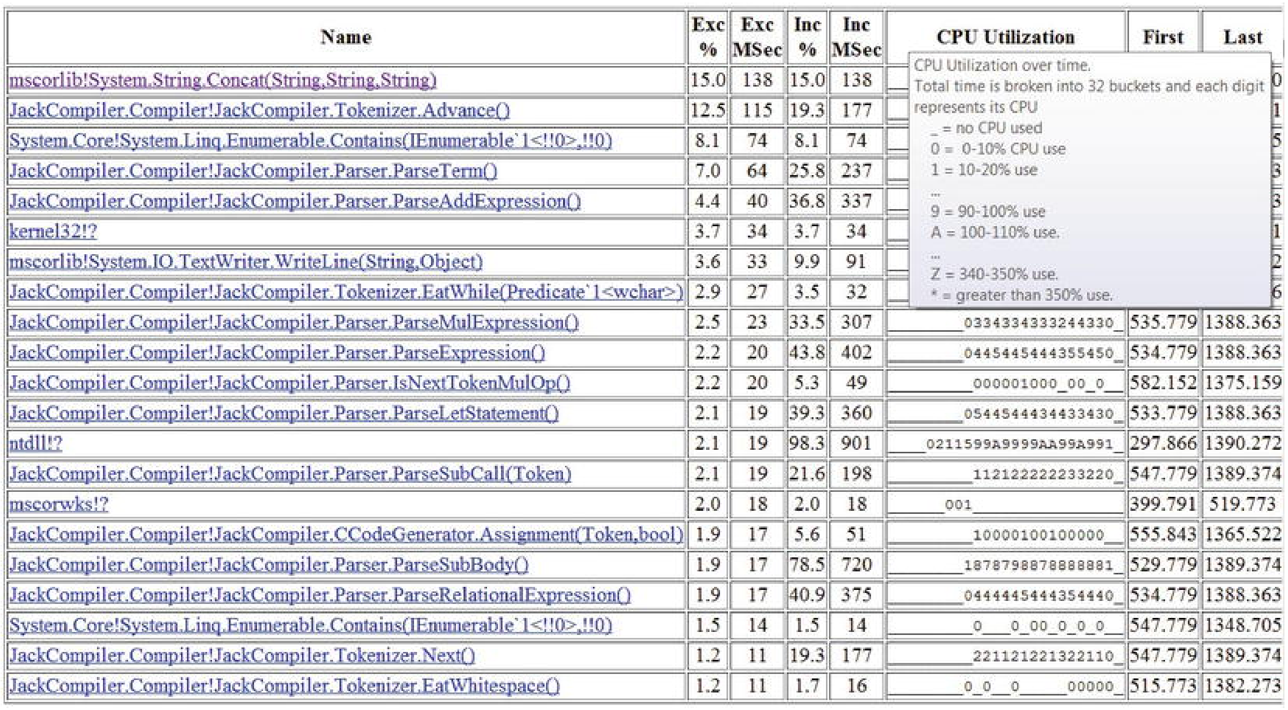
```
C:\PerfMonitor > perfmonitor runAnalyze JackCompiler.exe
Starting kernel tracing. Output file: PerfMonitorOutput.kernel.etl
Starting user model tracing. Output file: PerfMonitorOutput.etl
Starting at 4/7/2012 12:33:40 PM
Current Directory C:\PerfMonitor
Executing: JackCompiler.exe {
} Stopping at 4/7/2012 12:33:42 PM = 1.724 sec
Stopping tracing for sessions 'NT Kernel Logger' and 'PerfMonitorSession'.
```

```
Analyzing data in C:\PerfMonitor\PerfMonitorOutput.etlx
GC Time HTML Report in C:\PerfMonitor\PerfMonitorOutput.GCTime.html
JIT Time HTML Report in C:\PerfMonitor\PerfMonitorOutput.jitTime.html
Filtering to process JackCompiler (1372). Started at 1372.000 msec.
Filtering to Time region [0.000, 1391.346] msec
CPU Time HTML report in C:\PerfMonitor\PerfMonitorOutput.cpuTime.html
Filtering to process JackCompiler (1372). Started at 1372.000 msec.
Perf Analysis HTML report in C:\PerfMonitor\PerfMonitorOutput.analyze.html
PerfMonitor processing time: 7.172 secs.
```

The various HTML files generated by PerfMonitor contain the distilled report, but you can always use the raw ETL files with XPerfView or any other tool that can read binary ETW traces. The summary analysis for the example above contains the following information (this might vary, of course, when you run this experiment on your own machine):

- CPU Statistics—CPU time consumed was 917ms and the average CPU utilization was 56.6%. The rest of the time was spent waiting for something.
- GC Statistics—the total GC time was 20ms, the maximum GC heap size was 4.5MB, the maximum allocation rate was 1496.1MB/s, and the average GC pause was 0.1ms.
- JIT Compilation Statistics—159 methods were compiled at runtime by the JIT compiler, for a total of 30493 bytes of machine code.

Drilling down to the CPU, GC, and JIT reports can provide a wealth of useful information. The CPU detailed report provides information on methods that use a large percentage of CPU time (*bottom up analysis*), a call tree of where CPU time is spent (*top down analysis*), and individual caller-callee views for each method in the trace. To prevent the report from growing very large, methods that do not exceed a predefined relevance threshold (1% for the bottom up analysis and 5% for the top down analysis) are excluded. Figure 2-7 is an example of a bottom up report—the three methods with most CPU work were `System.String.Concat`, `JackCompiler.Tokenizer.Advance`, and `System.Linq.Enumerable.Contains`. Figure 2-8 is an example of (part of) a top down report—84.2% of the CPU time is consumed by `JackCompiler.Parser.Parse`, which calls out to `ParseClass`, `ParseSubDecls`, `ParseSubDecl`, `ParseSubBody`, and so on.



Name	Exc %	Exc MSec	Inc %	Inc MSec	CPU Utilization	First	Last
mscorlib!System.String.Concat(String,String,String)	15.0	138	15.0	138	[illegible]	[illegible]	[illegible]
JackCompiler.Compiler!JackCompiler.Tokenizer.Advance()	12.5	115	19.3	177	[illegible]	[illegible]	[illegible]
System.Core!System.Linq.Enumerable.Contains(IEnumerable`1<!!0>,!!0)	8.1	74	8.1	74	[illegible]	[illegible]	[illegible]
JackCompiler.Compiler!JackCompiler.Parser.ParseTerm()	7.0	64	25.8	237	[illegible]	[illegible]	[illegible]
JackCompiler.Compiler!JackCompiler.Parser.ParseAddExpression()	4.4	40	36.8	337	[illegible]	[illegible]	[illegible]
kernel32!?	3.7	34	3.7	34	[illegible]	[illegible]	[illegible]
mscorlib!System.IO.TextWriter.WriteLine(String,Object)	3.6	33	9.9	91	[illegible]	[illegible]	[illegible]
JackCompiler.Compiler!JackCompiler.Tokenizer.EatWhile(Predicate`1<wchar>)	2.9	27	3.5	32	[illegible]	[illegible]	[illegible]
JackCompiler.Compiler!JackCompiler.Parser.ParseMulExpression()	2.5	23	33.5	307	_______________0334334333244330_	535.779	1388.363
JackCompiler.Compiler!JackCompiler.Parser.ParseExpression()	2.2	20	43.8	402	_______________0445445444355450_	534.779	1388.363
JackCompiler.Compiler!JackCompiler.Parser.IsNextTokenMulOp()	2.2	20	5.3	49	________________000001000_00_0__	582.152	1375.159
JackCompiler.Compiler!JackCompiler.Parser.ParseLetStatement()	2.1	19	39.3	360	_______________0544544434433430_	533.779	1388.363
ntdll!?	2.1	19	98.3	901	___________0211599A9999AA99A991_	297.866	1390.272
JackCompiler.Compiler!JackCompiler.Parser.ParseSubCall(Token)	2.1	19	21.6	198	________________112122222233220_	547.779	1389.374
mscorwks!?	2.0	18	2.0	18	_________001____________________	399.791	519.773
JackCompiler.Compiler!JackCompiler.CCodeGenerator.Assignment(Token,bool)	1.9	17	5.6	51	________________10000100100000__	555.843	1365.522
JackCompiler.Compiler!JackCompiler.Parser.ParseSubBody()	1.9	17	78.5	720	_______________1878798878888881_	529.779	1389.374
JackCompiler.Compiler!JackCompiler.Parser.ParseRelationalExpression()	1.9	17	40.9	375	_______________0444445444354440_	534.779	1388.363
System.Core!System.Linq.Enumerable.Contains(IEnumerable`1<!!0>,!!0)	1.5	14	1.5	14	________________0___0_00_0_0_0__	547.779	1348.705
JackCompiler.Compiler!JackCompiler.Tokenizer.Next()	1.2	11	19.3	177	________________221121221322110_	547.779	1389.374
JackCompiler.Compiler!JackCompiler.Tokenizer.EatWhitespace()	1.2	11	1.7	16	_______________0_0__0_____00000_	515.773	1382.273

CPU Utilization over time.
Total time is broken into 32 buckets and each digit represents its CPU
_ = no CPU used
0 = 0-10% CPU use
1 = 10-20% use
...
9 = 90-100% use
A = 100-110% use.
...
Z = 340-350% use.
* = greater than 350% use.

Figure 2-7 . Bottom up report from PerfMonitor. The "Exc %" column is an estimation of the CPU time used by that method alone; the "Inc %" column is an estimation of the CPU time used by that method and all other methods it called (its sub-tree in the call tree)

Figure 2-8 . Top down report from PerfMonitor

The detailed GC analysis report contains a table with garbage collection statistics (counts, times) for each generation, as well as individual GC event information, including pause times, reclaimed memory, and many others. Some of this information will be extremely useful when we discuss the garbage collector's inner workings and performance implications in Chapter 4. Figure 2-9 shows a few rows for the individual GC events.

Figure 2-9 . Individual GC events, including the amount of memory reclaimed, the application pause time, the type of collection incurred, and other details

Finally, the detailed JIT analysis report shows how much time the JIT compiler required for each of the application's methods as well as the precise time at which they were compiled. This information can be useful to determine whether application startup performance can be improved—if an excessive amount of startup time is spent in the JIT compiler, pre-compiling your application's binaries (using NGen) may be a worthwhile optimization. We will discuss NGEN and other strategies for reducing application startup time in Chapter 10.

Tip Collecting information from multiple high-performance ETW providers can generate very large log files. For example, in default collection mode PerfMonitor routinely generates over 5MB of raw data per second. Leaving such a trace on for several days is likely to exhaust disk space even on large hard drives. Fortunately, both XPerf and PerfMonitor support circular logging mode, where only the last *N* megabytes of logs are retained. In PerfMonitor, the `/Circular` command-line switch takes the maximum log file size (in megabytes) and discards the oldest logs automatically when the threshold is exceeded.

Although PerfMonitor is a very powerful tool, its raw HTML reports and abundance of command-line options make it somewhat difficult to use. The next tool we'll see offers very similar functionality to PerfMonitor and can be used in the same scenarios, but has a much more user-friendly interface to collecting and interpreting ETW information and will make some performance investigations considerably shorter.

The PerfView Tool

PerfView is a free Microsoft tool that unifies ETW collection and analysis capabilities already available in PerfMonitor with heap analysis features that we will discuss later in conjunction with tools such as CLR Profiler and ANTS Memory Profiler. You can download PerfView from the Microsoft download center, at http://www.microsoft.com/download/en/details.aspx?id=28567. Note that you have to run PerfView as an administrator, because it requires access to the ETW infrastructure.

Figure 2-10 . PerfView's main UI. In the file view (on the left) a heap dump and an ETW trace are visible. The links on the main view lead to various commands the tool supports

To analyze ETW information from a specific process, use the Collect ➤ Run menu item in PerfView (Figure 2-10 shows the main UI). For the purpose of the heap analysis we will perform shortly, you can use PerfView on the MemoryLeak.exe sample application from this chapter's source code folder. It will run the process for you and generate a report with all the information PerfMonitor makes available and more, including:

- Raw list of ETW events collected from various providers (e.g. CLR contention information, native disk I/O, TCP packets, and hard page faults)
- Grouped stack locations where the application's CPU time was spent, including configurable filters and thresholds
- Stack locations for image (assembly) loads, disk I/O operations, and GC allocations (for every ~ 100KB of allocated objects)
- GC statistics and events, including the duration of each garbage collection and the amount of space reclaimed

Additionally, PerfView can be used to capture a heap snapshot from a currently running process or import a heap snapshot from a dump file. After importing, PerfView can be used to look for the types with most memory utilization in the snapshot and identify reference chains that are responsible for keeping these types alive. Figure 2-11 shows the PerfView reference analyzer for the `Schedule` class, which is responsible (inclusively) for 31MB of the heap snapshot's contents. PerfView successfully identifies the `Employee` class instances holding references to `Schedule` objects, while `Employee` instances are being retained by the f-reachable queue (discussed in Chapter 4).

Figure 2-11 . Reference chain for Schedule class instances, responsible for 99.5% of the application's memory usage in the captured heap snapshot

When we discuss memory profilers later in this chapter, we'll see that PerfView's visualization capabilities are still somewhat lacking compared to the commercial tools. Still, PerfView is a very useful free tool that can make many performance investigations significantly shorter. You can learn more about it using the built-in tutorial linked from its main screen, and there are also videos recorded by the BCL team that exhibit some of the tool's main features.

Custom ETW Providers

Similarly to performance counters, you might want to tap into the powerful instrumentation and information collection framework offered by ETW for your own application's needs. Prior to .NET 4.5, exposing ETW information from a managed application was fairly complex. You had to deal with plenty of details around defining a manifest for your application's ETW provider, instantiating it at runtime, and logging events. As of .NET 4.5, writing a custom ETW provider could hardly be easier. All you need to do is derive from the `System.Diagnostics.Tracing.EventSource` class and call the `WriteEvent` base class method to output ETW events. All the details of registering an ETW provider with the system and formatting the event data are handled automatically for you.

The following class is an example of an ETW provider in a managed application (the full program is available in this chapter's source code folder and you can run it with PerfMonitor later):

```
public class CustomEventSource : EventSource {
  public class Keywords {
   public const EventKeywords Loop = (EventKeywords)1;
   public const EventKeywords Method = (EventKeywords)2;
  }
  [Event(1, Level = EventLevel.Verbose, Keywords = Keywords.Loop,
   Message = "Loop {0} iteration {1}")]
  public void LoopIteration(string loopTitle, int iteration) {
   WriteEvent(1, loopTitle, iteration);
  }
  [Event(2, Level = EventLevel.Informational, Keywords = Keywords.Loop,
   Message = "Loop {0} done")]
  public void LoopDone(string loopTitle) {
   WriteEvent(2, loopTitle);
  }
  [Event(3, Level = EventLevel.Informational, Keywords = Keywords.Method,
   Message = "Method {0} done")]
  public void MethodDone([CallerMemberName] string methodName = null) {
   WriteEvent(3, methodName);
  }
}
class Program {
  static void Main(string[] args) {
   CustomEventSource log = new CustomEventSource();
   for (int i = 0; i < 10; ++i) {
   Thread.Sleep(50);
   log.LoopIteration("MainLoop", i);
   }
   log.LoopDone("MainLoop");
   Thread.Sleep(100);
   log.MethodDone();
  }
}
```

The PerfMonitor tool can be used to automatically obtain from this application the ETW provider it contains, run the application while monitoring that ETW provider, and generate a report of all ETW events the application submitted. For example:

```
C:\PerfMonitor > perfmonitor monitorDump Ch02.exe
Starting kernel tracing. Output file: PerfMonitorOutput.kernel.etl
Starting user model tracing. Output file: PerfMonitorOutput.etl
Found Provider CustomEventSource Guid ff6a40d2-5116-5555-675b-4468e821162e
Enabling provider ff6a40d2-5116-5555-675b-4468e821162e level: Verbose keywords: 0xffffffffffffffff
Starting at 4/7/2012 1:44:00 PM
Current Directory C:\PerfMonitor
Executing: Ch02.exe {

} Stopping at 4/7/2012 1:44:01 PM = 0.693 sec
Stopping tracing for sessions 'NT Kernel Logger' and 'PerfMonitorSession'.
Converting C:\PerfMonitor\PerfMonitorOutput.etlx to an XML file.
Output in C:\PerfMonitor\PerfMonitorOutput.dump.xml
PerfMonitor processing time: 1.886 secs.
```

Note There is another performance monitoring and system health instrumentation framework we haven't considered: *Windows Management Instrumentation* (WMI). WMI is a command-and-control (C&C) infrastructure integrated in Windows, and is outside the scope of this chapter. It can be used to obtain information about the system's state (such as the installed operating system, BIOS firmware, or free disk space), register for interesting events (such as process creation and termination), and invoke control methods that change the system state (such as create a network share or unload a driver). For more information about WMI, consult the MSDN documentation at http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582.aspx. If you are interested in developing managed WMI providers, Sasha Goldshtein's article "WMI Provider Extensions in .NET 3.5" (http://www.codeproject.com/Articles/25783/WMI-Provider-Extensions-in-NET-3-5, 2008) provides a good start.