

Username: Pralay Patoria **Book:** Pro .NET Performance. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Other Profilers

In this chapter, we chose to focus mostly on CPU, time, and memory profilers, because these are the metrics on which most performance investigations focus. There are several other performance metrics that have dedicated measurement tools; in this section we will briefly mention some of them.

Database and Data Access Profilers

Many managed applications are built around a database and spend a significant portion of their time waiting for it to return query results or complete bulk updates. Database access can be profiled at two locations: from the application's side, which is the realm of *data access profilers*, and from the database's side, best left to *database profilers*.

Database profilers often require vendor-specific expertise, and are typically used by database administrators in their performance investigations and routine work. We will not consider database profilers here; you can read more about the SQL Server Profiler, which is a very powerful database profiling tool for Microsoft SQL Server, at <http://msdn.microsoft.com/en-us/library/ms181091.aspx>.

Data access profilers, on the other hand, are well within the domain of application developers. These tools instrument the application's data access layer (DAL) and typically report the following:

- The database queries executed by the application's DAL, and the precise stack trace that initiated each operation.
- The list of application methods that initiated a database operation, and the list of queries that each method has run.
- Alerts for inefficient database accesses, such as performing queries with an unbounded result set, retrieving all table columns while using only some of them, issuing a query with too many joins, or making one query for an entity with *N* associated entities and then another query for each associated entity (also known as the "SELECT N + 1" problem).

There are several commercial tools that can profile application data access patterns. Some of them work only with specific database products (such as Microsoft SQL Server), while others work with only a specific data access framework (such as Entity Framework or NHibernate). Below are a few examples:

- RedGate ANTS Performance Profiler can profile application queries to a Microsoft SQL Server database.
- Visual Studio "Tier Interactions" profiling feature can profile any synchronous data access operation from ADO.NET—unfortunately, it does not report call stacks for database operations.
- The Hibernating Rhinos family of profilers (LINQ to SQL Profiler, Entity Framework Profiler, and NHibernate Profiler) can profile all operations performed by a specific data access framework.

We will not discuss these profilers in more detail here, but if you are concerned about the performance of your data access layer, you should consider running them alongside with a time or memory profiler in your performance investigations.

Concurrency Profilers

The rising popularity of parallel programming warrants specialized profilers for highly concurrent software that uses multiple threads to run on multiple processors. In [Chapter 6](#) we will examine several scenarios in which easily ripe performance gains are available from parallelization—and these performance gains are best realized with an accurate measurement tool.

The Visual Studio profiler in its Concurrency and Concurrency Visualizer modes uses ETW to monitor the performance of concurrent applications and report several useful views that facilitate detecting scalability and performance bottlenecks specific to highly concurrent software. It has two modes of operation demonstrated below.

Concurrency mode (or *resource contention profiling*) detects resources, such as managed locks, on which application threads are waiting. One part of the report focuses on the resources themselves, and the threads that were blocked waiting for them—this helps find and eliminate scalability bottlenecks (see [Figure 2-32](#)). Another part of the report displays contention information for a specific thread, i.e. the various synchronization mechanisms for which the thread had to wait—this helps reduce obstacles in a specific thread's execution. To launch the


profiler in this mode of operation, use the Performance Explorer pane or Analyze  Launch Performance Wizard menu item and select the Concurrency mode.

Figure 2-32. Contention for a specific resource—there are several threads waiting at once to acquire the resource. When a thread is selected, its blocking call stack is listed on the bottom

Concurrency Visualizer mode (or *multithreaded execution visualization*) displays a graph with the execution details of all application threads, color-coded according to their current state. Every thread state transition—blocking for I/O, waiting for a synchronization mechanism, running—is recorded, along with its call stack and unblocking call stack when applicable (see [Figure 2-33](#)). These reports are very helpful for understanding the role of application threads, and detecting poor performance patterns such as oversubscription, undersubscription, starvation, and excessive synchronization. There's also built-in support in the graph for Task Parallel Library mechanisms such as parallel loops, and CLR


synchronization mechanisms. To launch the profiler in this mode of operation, use the Analyze  Concurrency Visualizer sub-menu.

Figure 2-33. Visualization of several application threads (listed on the left) and their execution. From the visualization and the histogram on the bottom it is evident that work was not distributed evenly between the different threads

Note MSDN features a collection of anti-patterns for multithreaded applications based on Concurrency Visualizer graphs, including lock convoy, uneven workload distribution, oversubscription, and others—you can find these anti-patterns online at [http://msdn.microsoft.com/en-us/library/ee329530\(v=s.110\).aspx](http://msdn.microsoft.com/en-us/library/ee329530(v=s.110).aspx). When you run your own measurements, you'll be able to identify similar problems by visually comparing the reports.

Concurrency profiling and visualization are highly useful tools, and we will meet them again in subsequent chapters. They serve as another great evidence of ETW's great influence—this omnipresent high-performance monitoring framework is used across managed and native profiling tools.

I/O Profilers

The last performance metric category we study in this chapter is I/O operations. ETW events can be used to obtain counts and details for physical disk access, page faults, network packets, and other types of I/O, but we haven't seen any specialized treatment for I/O operations.

Sysinternals Process Monitor is a free tool that collects file system, registry, and network activity (see [Figure 2-34](#)). You can download the entire suite of Sysinternals tools, or just the latest version of Process Monitor, from the TechNet website at <http://technet.microsoft.com/en-us/sysinternals/bb896645>. By applying its rich filtering capabilities, system administrators and performance investigators use Process Monitor to diagnose I/O-related errors (such as missing files or insufficient permissions) as well as performance problems (such as remote file system accesses or excessive paging).

Figure 2-34. Process Monitor showing several types of events in the main view and the call stack for a specific event in the dialog window. Frames 19 and lower are managed frames

Process Monitor offers a complete user-mode and kernel-mode stack trace for each event it captures, making it ideal to understand where excessive or erroneous I/O operations are originating in the application's source code. Unfortunately, at the time of writing Process Monitor cannot decode managed call stacks—but it can at least point in the general direction of the application that performed the I/O operation.

Through the course of this chapter we used automatic tools that measure application performance from various aspects—execution time, CPU time, memory allocation, and even I/O operations. The variety of measurement techniques is overwhelming, which is one of the reasons why developers often like to perform manual benchmarks of their application's performance. Before concluding this chapter, we discuss microbenchmarking and some of its potential pitfalls.