Most of this book focuses on optimizing computational aspects of application performance. We have seen numerous examples, such as tuning garbage collection, parallelizing loops and recursive algorithms, and even by coming up with better algorithms to reduce runtime costs. For some applications, optimizing only the computational aspect results in limited performance gains, because the performance bottleneck lies in I/O work, such as network transfers or disk accesses. In our experience, a considerable portion of performance problems encountered in the field is not caused by an unoptimized algorithm or excessive CPU utilization, but is due to an inefficient utilization of the system’s I/O devices. Let us consider two scenarios in which optimizing I/O can result in performance gains:

- An application might incur a significant computational (CPU) overhead due to inefficient use of I/O, which comes at the expense of useful work. Worse, this overhead might be so high that it becomes the limiting factor to realizing the full potential capacity of the I/O device.
- The I/O device might be under-utilized or its capacity is wasted because of inefficient usage patterns, such as making many small I/O transfers or by failing to keep the channel fully utilized.

This chapter discusses strategies for improving I/O performance in general and network I/O performance in particular. In addition, we cover serialization performance and compare several serializers.

General I/O Concepts
This section explores I/O concepts and provides performance guidelines pertaining to I/O of any kind. This advice is applicable to networking applications, heavy disk-accessing processes, and even software designed to access a custom high-bandwidth hardware device.

Synchronous and Asynchronous I/O
With synchronous I/O, the I/O transfer function (e.g. ReadFile, WriteFile, or DeviceIoControl Win32 API functions) blocks until the I/O operation completes. Although this model is convenient to use, it is not very efficient. During the time between issuing successive I/O requests, the device may be idle and, therefore, is potentially under-utilized. Another problem with synchronous I/O is that a thread is “wasted” for each concurrently pending I/O request. For example, in a server application servicing many clients concurrently, you may end up creating a thread per session. These threads, which are essentially mostly idle, are wasting memory...
and may create a situation called *thread thrashing* in which many threads wake up when I/O completes and compete with each other for CPU time, resulting in many context switches and poor scalability.

The Windows I/O subsystem (including device drivers) is internally asynchronous – execution of program flow can continue while an I/O operation is in progress. Almost all modern hardware is similarly asynchronous in nature as well and does not need polling to transfer data or to determine if an I/O operation is complete. Most devices instead rely on Direct Memory Access (DMA) controllers to transfer data between the device and the computer RAM, without requiring the CPU’s attention during the transfer, and then raise an interrupt to signal completion of the data transfer. It is only at the application level that Windows allows synchronous I/O that is actually asynchronous internally.

In Win32, asynchronous I/O is called *overlapped I/O* (see Figure 7-1 comparing synchronous and overlapped I/O). Once an application issues an overlapped I/O, Windows either completes the I/O operation immediately or returns a status code indicating the I/O operation is still pending. The thread can then issue more I/O operations, or it can do some computational work. The programmer has several options for receiving a notification about the I/O operation’s completion:

- Signaling of a Win32 event: A wait operation on this event will complete when the I/O completes.
- Invocation of a user callback routine via the Asynchronous Procedure Call (APC) mechanism: The issuing thread must be in a state of *alertable wait* to allow APCs.
- Notification via I/O Completion Ports: This is usually the most efficient mechanism. We explore I/O completion ports in detail later in this chapter.

**Figure 7-1.** Comparison between synchronous and overlapped I/O

---

**Note** Some I/O devices (e.g. a file opened in unbuffered mode) benefit (by increasing device utilization) if an application can keep a small amount of I/O requests pending. A recommended strategy is to pre-issue a certain number of I/O requests and, for each request that completes, re-issue another. This ensures the device driver can initiate the next I/O as quickly as possible, without waiting for the application to issue the next I/O in response. However, do not exaggerate with the amount of pending data, since it can consume limited kernel memory resources.