Learning .NET High-performance Programming

This book will help you understand what "programming for performance" means, and use effective coding patterns and techniques to optimize your .NET applications. You will begin by understanding what "high performance coding" means, and the different performance concerns. You will see how CLR works and get an understanding of concepts such as memory management, garbage collection, and thread life cycles. You will proceed to learn about the theoretical and practical concepts of PLINQ programming. You will also see what Big Data is, and how to architect a Big Data solution to manipulate large datasets. Finally, you will learn how to launch and analyze a profile session and execute tests against a code block or application for performance analysis.

By the end of this book, you will have a complete understanding of efficient programming using high-performance techniques, and will be able to write highly optimized applications.

Who this book is written for

If you are a .NET developer with an understanding of application development, but want to learn how to optimize the performance of your applications, this is the book for you. Basic knowledge of C# is expected.

What you will learn from this book

- Understand the significance of high performance in applications
- Identify different performance concerns for all the main used architectures and designs
- Acquaint yourself with the commonly used MVC and MVVM patterns
- Understand the fundamentals of CLR
- Learn more about Task Parallel Library and concepts such as Thread Pool Tuning and parameter injection
- Use PLINQ programming to achieve parallelism
- Design Big Data solutions for handling datasets optimally
- Choose high-performance querying strategies to retrieve and manipulate data
- Execute tests on applications for performance analysis

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Performance Thoughts'
- A synopsis of the book’s content
- More information on Learning .NET High-performance Programming
About the Author

Antonio Esposito has been an experienced software developer, architect, and trainer since 2002. He started programming in BASIC on MS-DOS 3 at the age of 8 and still loves programming on Microsoft products. An expert in .NET, he has a lot of knowledge of SOA techniques and architectures as well as WCF and Microsoft Azure. He also has extensive knowledge of application production frameworks, such as WPF, MVC, and so on.

Antonio has worked with prominent companies as a freelance consultant, speaker, and trainer. These companies include UniCredit Banca, Microsoft Italia, IBM Italia, Ferrari, Tenova Pomini, Cornér Banca, and many others.

As a speaker, he has had the opportunity to contribute to events such as the MCT Summit Europe, WPC Italia, Community Days Milano, Codemotion, and many more from his own user group, DotNet Lombardia.
For most of us, "performance" is a word with a single meaning. When it comes to software production, this meaning usually corresponds to something fast. A fast program is a good program. Although this is not a wrong assertion, the meaning of the word "performance" is wider and deeper.

Writing a responsive UI actually means writing a performing UI. Deploying a worker role across Microsoft Azure, which is able to scale up to 100 cores, and handling millions of messages per hour actually means writing a performing workflow. The two examples show two different kinds of performance, and more exist.

Other than multiple meanings, the word "performance" also refers to multiple implementation levels. For example, a developer has to keep the security aspect of his application in mind right from the outset, because using a simple X509 certificate does not make an insecure web application secure. The same is true when it comes to performance.

If we want to create a high-performance application, we have to design a high-performance architecture right from the start, implement performance-oriented strategies, and bring up a good performance-engineered project that is able to assist the wider development project to create valid performance requisites and tests.

As developers, we cannot avoid mastering all the techniques required to help us face day-to-day challenges. Asynchronous programming and parallel programming are two examples. Mastering such techniques helps us create good software in terms of responsiveness and scalability. A clear understanding of the .NET Framework has to become part of the knowledge arsenal for any .NET developer; understanding memory management, process isolation, and thread life cycle are examples.

Lots of real-world examples will be available in this book. They will be about the most widely used programming techniques and scenarios, together with special-case programming recipes for big data, engineering, and database integration.

All of these topics are covered here with enthusiasm and expertise.
What this book covers

Chapter 1, *Performance Thoughts*, gives you an overview of the term "performance" and related concepts.


Chapter 3, *CLR Internals*, gives you in-depth knowledge of the internals of CLR, from memory management to thread life cycle management.

Chapter 4, *Asynchronous Programming*, equips you with the ability to program methods that will never keep the user waiting.

Chapter 5, *Programming for Parallelism*, gives you many details of parallelism, and covers scenarios such as this one: a lot of data items are available for our business logic. Mastering parallelism techniques brings any application throughput to extremely high levels, with little effort by the developer.

Chapter 6, *Programming for Math and Engineering*, shows you real-world examples for specific cases related to scientific elaboration.

Chapter 7, *Database Querying*, demonstrates working with databases. Any business-related application or enterprise application deals with a lot of data. This chapter also provides good knowledge of all persistence-layer access frameworks and techniques that make such accesses fast and reliable for all kinds of applications.

Chapter 8, *Programming for Big Data*, covers working with huge and fast-growing datasets that have specific issues and difficulties, with a simplified presentation and a lot of examples.

Chapter 9, *Analyzing Code Performance*, proves that good programming and design are not enough if we don't always monitor our software performance results in design time and runtime.
In software engineering, the most misused word is performance. Although anyone may like a performing application or website, the word itself hides a lot of meanings, each with specific pros and cons.

A professional programmer must have a deep understanding of the various facets of the term performance, as the term assumes different meanings in different scenarios.

A well-performing application should comply with different kinds of performance requirements, which usually change according to the application's architecture and design. It should also focus on the market expectations and (sometimes) what the current development trend is.

As C# programmers, we must add to the generic knowledge about performance-oriented programming. All these skills let us achieve the best results from coding. Choosing the best architecture solution or design pattern will give a boost to long- or short-term performance results (explained later in this chapter). However, implementing these architectures with the wrong design, will nullify expectations of speed or quality we planned. This chapter will guide you on the meanings and facets of the term performance, as implied when programming for Microsoft .NET Framework:

- Understanding performance
- Performance as a requirement
- Performance engineering
- Performance aspects
- Class of applications
- Technical overview
Performance Thoughts

Understanding performance
When you talk about performance with respect to the results of an application being developed, it is a word that means good results for the given expectations.

Without diving into the details of the meaning, it is clear that the keyword here is not the search for good results but the comparison between those results with a specific reference value. No static, relative, or ranged value can have any significance without some kind of a legend associated with it.

Diving into the meaning of the phrase good results for the given expectations, there is another hidden important key concept: the availability to measure, in technical terms, any given aspect of our application. Such terms must be numerically defined, such as a time, a size expressed in Bytes (or multiples), and so on.

In other words, performance is associated with all the measurable aspects of an application.

As software developers, we have to understand client needs. We cannot be simply code writers or technical enthusiasts.

Although we have to mine technical requisites between use cases and user expectations, we have to guide the client to give us useful information regarding their expectations. I know they do not know anything about software engineering, but it is up to us to let them learn at least the basics here. Sometimes, we have to mine requisites by ourselves, while other times we can try to get the client to use the right requisite formula with suggestions, questions, and indications.

Any requisite with no relative or absolute reference value exposed is invalid.

Subtle to define as not valid is any performance requisite with some generic numeric needs, without specifying any context or value legend. An example will be a request like a web page response time, without specifying the server load or page computational complexity.

Taking some time to reflect on what we just read, we found another aspect of the term performance, which is a technical value and may become a performance indicator only if it's compared to a valid expected range.

Let's evaluate another client need. A client asks for a web page to be able to respond in less than 1 second in a low load time window (less than 1,000 active users) or not more than 10 seconds with heavy load equal to or more than 10,000 active users.

Here, we do have a valid request against a value range and environment, but there is still something missing, such as a shared and documented test case, which acts as a reference to everyone working on the project.
Chapter 1

An example on a valid client's need for performance requirement would be that a client asks for a web application to execute Test001 in less than one second with less than 1.000 active users online, or be able to execute the same test case in less than 10 seconds with no more than 10.000 active online users.

**Performance as a requirement**

Talking about performance as a need for a client (or buyer), it is easy to infer how this is definitely a requirement and not a simple need that a client may have.

In software engineering, the requirement collection (and analysis) is called Requirements engineering. We found a couple of specific requirements that we should properly understand as software developers: the functional and non-functional requirements.

Under the functional requirement, we can find what a software must do, and this (and other specifications) codes what we call software design. While with a non-functional requirement, we focus on how the system (and hence, the software) has to work. This (and other specifications) codes what we call system architecture.

In other words, when a client asks for an application to compute something (and if the computation hides a proprietary logic, the formula is part of the requirement as a technical detail), they are asking for a function, so this is a functional requirement.

When a client asks for an application to work only if authenticated (a non-functional requirement), they are definitely asking that an application works in a specific manner, without asking the application to produce a target or goal in the results.

Usually, anything about security, reliability, testability, maintainability, interoperability, and performance guidelines, are all non-functional requirements. When the client asks the software what needs can be satisfied with respect to their business, it is actually a functional requirement.

Although a client may ask for a fast or responsive application, they are not actually asking for something related to their business or what to do with the software, they are simply asking for some generic feature; in other words, a wish. All such technical wishes are non-functional requirements. But what about a scenario in which the client asks for something that is a business requirement?

Let's say that a client asks for an application to integrate with a specific industrial bus that must respond in less than 100 milliseconds to any request made throughout the bus. This now becomes a functional requirement. Although this is not related to their business, logically, this is a technical detail related to their domain, and has become a proper functional (domain-related) requisite.
Performance engineering

Performance engineering is the structure behind the goal to succeed in respecting all the nonfunctional requirements that a software development team should respect.

In a structured software house (or enterprise), the performance engineering is within system engineering, with specific roles, skills, tools, and protocols.

The goal here is not only to ensure the availability of the expected performance requirements during the development stage, but also how these requirements evolve when the application evolves, and its lifecycle up to the production environment, when continuous monitoring of the current performance against the initial requirements gives us a direct and long-range analysis of the system running.

We live in a time when an IT team is definitely an asset for most companies. Although there are still some companies that don’t completely understand the definition of IT and think of it as an unnecessary cost, they will at least see the importance of performance and security as the most easily recognizable indicators of a well-made application.

Performance engineering has objectives that cover the easy goal of how to write a fast application. Let’s take a look at some of these objectives, as follows:

1. Reducing software maintenance costs.
2. Increasing business revenue.
3. Reducing hardware acquisition costs.
4. Reducing system rework for performance issues.

Here, the focus is on all aspects of software development that good performance engineering may optimize. It is obvious that a more powerful application leads to lesser hardware requirements, although it is still obvious that a well-made application needs less reworks for performance issues. The focus is not on the time or money saved, but the importance of thinking about performance from the beginning of the development project up to the production stage. Writing a performing piece of code is an easy task compared to a complete software development project, with performance in mind. I know that coding is loved by any developer, but as a professional, we have to do something more.
Reducing the work to fix issues and the cost of having developers working on performance optimization or system tuning after an application is deployed in the production stage enforces the contract with the client/buyer who commissioned the software development. This respects the performance requisites and builds trust with the customer as well as leading to a sensible reduction in maintenance costs.

In performance engineering, a formal performance requisite is coded at the beginning of the development stage, together with software and system architects. Multiple tests are then executed during the development lifecycle in order to satisfy requisites (first) and maintain the level of success at the time. At the end of the production stage, the performance test analysis will act as proof of the work done in programming, testing, releasing, and maintaining of the software, as well as an indicator for various kind of issues not related directly to performance (a disk failure, a DoS instance, a network issue, and so on).

**Performance aspects**

When working on performance requirements, in the development stage or for the complete application lifecycle, we have to choose the performance aspects that influence our software and development project. Before writing the code, many decisions will be taken, such as what architecture the software must have, what design to implement, what hardware target will run our software, and so on.

As said previously, anything technically measurable and comparable with a valid value range may become a performance indicator and therefore a performance requirement. The more this indicator becomes specific to the application that is being developed, the more its requirements becomes domain related, while for all the others, they are generic non-functional requirements.

We have to always keep in mind that a lot of things may become performance indicators from the technical standpoint, such as the ability to support multithreading or parallel programming, also system-specific indicators, such as the ability to support multicore or a specific GPU's programming languages, but these are only details of a well-formed performance requisite.
A complete performance requisite usually covers multiple aspects of performance. Many aspects do exist. Think of this requirement as a map, as follows:

<table>
<thead>
<tr>
<th>Aspect</th>
<th>0%</th>
<th>20%</th>
<th>40%</th>
<th>60%</th>
<th>80%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource usage</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

A performance aspect map is a simple grid, exposing the importance of performance aspects

The first thing to keep in mind is that we cannot have every aspect that is shown in the preceding figure as our primary performance goal. It is simply impossible for hardware and software reasons. Therefore, the tricky task here is to find the primary goal and every secondary or less important objective that our application needs to satisfy. Without any regret, some aspect may become completely unnecessary for our application. Later in this chapter, we will cover a few test cases.

Putting extreme focus on a single kind of performance may lead to a bad performing application.

A desktop or mobile application will never scale out, so why focus on it? A workflow never interacts directly with a client; it will always work in an asynchronous way, so why focus on latency? Do not hesitate to leave some of this aspect in favor of other, more critical aspects.

Let's look at the most important and widely recognized performance aspects.

**Latency**

The latency is the time between a request and response, or more specifically, the time between any action and its result. In other words, *latency is the time between a cause and its effect*, such that a user can feel it.

A simple example of latency issues is someone using an RDP session. What lets us feel that we are using an RDP session is the latency that the network communication adds to the usual keyboard and mouse iteration.
Latency is critical in web applications where any round-trip between the client's browser and server and then back to the browser is one of the main indicators about the website's responsiveness.

**Throughput**

One of the most misused words, a synonym for power, or for the most part, the synonym for good programming, is throughput. Throughput simply means that the speed rate of anything is the main task of the given product or function being valued.

For instance, when we talk about an HDD, we should focus on a performance indicator to reassume all the aspects of HDD speed. We cannot use the sequential read/write speed, and we cannot use the seek time as the only indicator to produce a throughput valuation. These are specific performance indicators of the domain of HDD producers. The following guidelines are also mentioned at the beginning of the chapter. We should find a good indicator (direct, indirect, or interpolated) to reassume the speed of the HDD in the real world. Is this what a performance test suite does for a system and HDD? We can use a generic random 64K read/write (50/50 percent) test to produce a single throughput indicator.

Talking about software, the ability of a workflow to process transactions in a timely manner (such as per second or per hour) is another valid throughput performance indicator.

**Resource usage**

This is another key performance indicator that includes everything about resource usage such as memory, CPU, or GPU (when applicable).

When we talk about resource usage, the primary concern is memory usage. Not because the CPU or GPU usage is less important, but simply because the GPU is a very specific indicator, and the CPU usually links to other indicators such as throughput.

The GPU indicator may become important only if the graphical computation power is of primary importance, such as when programming for a computer game. In this case, the GPU power consumption becomes a domain-specific (of game programming) technical indicator.

A memory leak may occur when the memory is partially or totally unreleased within a process, when unused.
Performance Thoughts

That being said, it is easy to infer that for the resource usage indicator, the most important feature is memory consumption. If we need to load a lot of data together (in the following chapters, we will see alternatives to this solution), we will have to set up hardware resources as needed.

If our application never releases unused memory, we will face a memory leak. Such a leak is a tremendous danger for any application. OutOfMemoryException is an exception, which in the .NET programming world means that no more memory is available to instantiate new objects.

The only chance to find a memory leak is by profiling the entire application with a proper tool (we will see the integrated profiling tool of Visual Studio in Chapter 9) to show us how an application consumes memory on a subroutine basis.

Availability/reliability

This is a key performance indicator for any software serving multiple users, such as a web service, web application, workflow, and so on.

Availability is also the proof of how a performance indicator may also be something not directly related to speed or power, but simply the ability of the software being in up-time, actually running, without issues in any condition. Availability is directly related to reliability. The more a system is available, the more it is reliable. However, a system may become available using a good maintenance plan or a lot of rework. A reliable system is always a strong one that does not need special maintenance or rework because it was well developed at the beginning, and meets most of the challenges that the production stage can produce.

Scalability

When talking about scalability, things come back to some kind of power—the ability of a single function or entire application to boost its performance—as the number of processors rise or the number of servers increases. We will focus a lot on this indicator by searching for good programming techniques such as multithreading and parallel programming in this and the following chapters, because at the time of writing this book, CPU producers have abandoned the path of single processor power, in favor of multicore CPU architectures. Today, we see smartphones with a CPU of four cores and servers with a single socket of twenty cores each. As software developers, we have to follow market changes, and change our software accordingly to take the most advantages possible.
Scalability is not too difficult to achieve because of the great availability of technologies and frameworks. However, it is not something we can always achieve and at any level. We can neither rely only on hardware evolution, nor on infinite scalability, because not all our code maybe scalable. If they are, it is always limited by the technology, the system architecture, or the hardware itself.

**Efficiency**

Efficiency is a relatively new kind of performance indicator. The existence of mobile devices and computer-like laptops since 1975, with the release of IBM 5100, opened the way to a new performance indicator of efficiency. Absolute power consumption is a part of the meaning of efficiency, with a new technical indicator named *performance per watt*, an indicator that shows the computation level that consumes a single watt of power.

As software developers, we will never focus on hardware electrical consumption, but we have to reduce, at the most, any overhead. Our goal is to avoid wasting any computational power and consequently, electrical power. This aspect is critical in mobile computing, where battery life is never enough.

Speaking of cloud computing, efficiency is a critical indicator for the cloud provider that sells the virtual machines in a time-based billing method, trying to push as many billable VMs in the same hardware. Instead, for a cloud consumer, although efficiency is something outside of their domain, wasting CPU power will force the need to use more VMs. The disadvantage of this is to pay more to have the same results.

In my opinion, always take into consideration this aspect, at least a bit, if you want to reduce global electrical consumption.

**Class of applications**

The performance requirement analysis is not easy to obtain.

A lot of aspects actually exist. As explained at the beginning of the last paragraph, we have to strike the right balance between all performance aspects, and try to find the best for our target application.

Trying to get the best from all the aspects of performance is like asking for no one at all, with the added costs of wasting time in doing something that is not useful. It is simply impossible reaching the best for all aspects all together. Trying to obtain the best from a single aspect will also give a bad overall performance. We always must make a priority table like the aspect map already seen in preceding paragraphs.
Different types of applications have different performance objectives, usually the same per type. Here are some case studies for the three main environments, namely desktop, mobile, and server-side applications.

**Case study: performance aspects of a desktop application**

The first question we should ask ourselves, when designing the performance requirements of a desktop class application, is *to whom is this application going to serve?*

A desktop class application serves a single user per system.

Although this is a single-user application, and we will never need scalability at the desktop level, we should consider that the architecture being analyzed has a perfect scalability by itself.

For each new user using our application, a new desktop will exist, so new computational power will be made available to users of this application. Therefore, we can assume that scalability is not a need in the performance requisite list of this application kind. Instead, any server being contacted by this kind of application will become a bottleneck if it is unable to keep up with the increasing demands.

As written by Mr. Jakob Nielsen in 1993, a usability engineer, human users react as explained in the following bullet list:

- 100 milliseconds is the time limit to make sure an application is actually reacting well
- 1 second is the time limit to bring users to the application workflow, otherwise users will experience delay
- 10 seconds is the time limit to keep the users' attention on the given application

It is easy to understand that the main performance aspect composing a requisite for a desktop application is latency.

Low resource usage is another key aspect for a desktop application performance requisite because of the increasingly smaller form factor of mobile computing, such as the Intel Ultrabook®, device with less memory availability. The same goes for efficiency.

It is strange to admit that we do not need power, but this is the truth because a single desktop application is used by a single user, and it is usually unable to fulfil the power resources of a single desktop class system.
Another secondary goal for this kind of performance requirement is availability. If a single application crashes, this halts the users productivity and in turn might lead to newer issues such that, the development team will need to fix it. This crash affects only a single user, leaving other user application instances free by any kind of related issues.

Something that does not impact a desktop class application, as explained previously, is scalability, because multiple users will never be able to use the same personal computer all together.

This is the target aspect map for a desktop class application:

<table>
<thead>
<tr>
<th>Latency</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0% 20% 40% 60% 80% 100%</td>
</tr>
<tr>
<td>Latency</td>
<td>X</td>
</tr>
<tr>
<td>Throughput</td>
<td>X</td>
</tr>
<tr>
<td>Resource usage</td>
<td>X</td>
</tr>
<tr>
<td>Availability</td>
<td>X</td>
</tr>
<tr>
<td>Scalability</td>
<td>X</td>
</tr>
<tr>
<td>Efficiency</td>
<td>X</td>
</tr>
</tbody>
</table>

The aspect map of a desktop application relying primary on a responding UI

**Case study: performance aspects of a mobile application**

When developing a mobile device application, such as for a smartphone device or tablet device, the key performance aspect is resource usage, just after Latency.

Although a mobile device application is similar to a desktop class one, the main performance aspect here is not latency because on a small device with (specifically for a Modern UI application) an asynchronous programming model, latency is something overshadowed by the system architecture.
Performance Thoughts

This is the target aspect map for a mobile device application:

<table>
<thead>
<tr>
<th>Latency</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0% 20% 40% 60% 80% 100%</td>
</tr>
<tr>
<td>Latency</td>
<td>X</td>
</tr>
<tr>
<td>Throughput</td>
<td>X</td>
</tr>
<tr>
<td>Resource usage</td>
<td>X</td>
</tr>
<tr>
<td>Availability</td>
<td>X</td>
</tr>
<tr>
<td>Scalability</td>
<td>X</td>
</tr>
<tr>
<td>Efficiency</td>
<td>X</td>
</tr>
</tbody>
</table>

The aspect map of a mobile application relying primarily on low resource usage.

Case study: performance aspects of a server application

When talking about a server-side application, such as a workflow running in a completely asynchronous scenario or some kind of task scheduler, things become so different from the desktop and mobile device classes of software and requirements.

Here, the focus is on throughput. The ability to process as many transactions the workflow or scheduler can process.

Things like Latency are not very useful because of the missing user interaction. Maybe a good state machine programming may give some feedback on the workflow status (if multiple processing steps occur), but this is beyond the scope of the Latency requirement.

Resource usage is also sensible here because of the damage a server crash may produce. Consider that the resource usage has to multiply for the number of instances of the workflow actually running in order to make a valid estimation of the total resource usage occurring on the server. Availability is part of the system architecture if we use multiple servers working together on the same pending job queue, and we should always make this choice if applicable, but programming for multiple asynchronous workflow instances may be tricky and we have to know how to avoid making design issues that can break the system when a high load of work comes. In the next chapter, we will look at architectures and technologies we can use to write a good asynchronous and multithreaded code.
Let's see my aspect map for the server-side application class, shown as follows:

<table>
<thead>
<tr>
<th>Latency</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0% 20% 40% 60% 80% 100%</td>
</tr>
<tr>
<td>Latency</td>
<td>X</td>
</tr>
<tr>
<td>Throughput</td>
<td>X</td>
</tr>
<tr>
<td>Resource usage</td>
<td>X</td>
</tr>
<tr>
<td>Availability</td>
<td>X</td>
</tr>
<tr>
<td>Scalability</td>
<td>X</td>
</tr>
<tr>
<td>Efficiency</td>
<td>X</td>
</tr>
</tbody>
</table>

The aspect map of a server-side application relying primary on high processing speed

When dealing with server-side applications that are directly connected to user actions, such as a web service responding to a desktop application, we need high computation power and scalability in order to respond to requests from all users in a timely manner. Therefore, we primarily need low latency response, as the client is connected (also consuming resources on the server), waiting for the result. We need availability because one or more application depends on this service, and we need scalability because users can grow up in a short time and fall back in the same short time. Because of the intrinsic distributed architecture of any web service-based system, a low resource usage is a primary concern; otherwise, the scalability will never be enough:

<table>
<thead>
<tr>
<th>Latency</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0% 20% 40% 60% 80% 100%</td>
</tr>
<tr>
<td>Latency</td>
<td>X</td>
</tr>
<tr>
<td>Throughput</td>
<td>X</td>
</tr>
<tr>
<td>Resource usage</td>
<td>X</td>
</tr>
<tr>
<td>Availability</td>
<td>X</td>
</tr>
<tr>
<td>Scalability</td>
<td>X</td>
</tr>
<tr>
<td>Efficiency</td>
<td>X</td>
</tr>
</tbody>
</table>

A user invoked server-side application aspect-map relying primary on latency speed

The aspect map of a server-side web service-based application carefully uses cloud-computing auto-scale features. Scaling out can help us in servicing thousands of clients with the right number of VMs. However, in cloud computing, VMs are billable, so never rely only on scalability.
It is not necessary to split the aspects trying to cover each level of magnitude, but it is a good practice to show the precedence order.

Performance concerns as time changes
During the lifecycle of an application living in the production stage, it may so happen that the provisioned performance requisite changes.

The more focus we put at the beginning of the development stage in trying to fulfil any future performance needs, the less work we will need to do to fix or maintain our application, once in the production stage.

The most dangerous mistake a developer can make is underestimate the usage of a new application. As explained at the beginning of the chapter, performance engineering is something that a developer must take care of for the entire duration of the project. What if the requirement used for the duration of the development stage is wrong when applied to the production stage? Well, there is not much time to recriminate. Luckily, software changes are less dangerous than hardware changes. First, create a new performance requirement, and then make all brand new test cases that can be applied to the new requirements and try to execute this on the application as in the staging environment. The result will give us the distance from the goal! Now, we should try to change our code with respect to the new requirements and test it again. Repeating these two steps until the result becomes valid against the given value ranges.

Talking, for instance, about a desktop application, we just found that the ideal aspect map focuses a lot on the responsiveness given by low Latency in user interaction. If we were in 2003, the ideal desktop application in the .NET world would have been made on Windows Forms. Here, working a lot with technologies such as Thread Pool threads would help us achieve the goal of a complete asynchronous programming to read/write any data from any kind of system, such as a DB or filesystem, thus achieving the primary goal of a responsive user experience. In 2005, a BackgroundWorker class/component could have done the same job for us using an easier approach. As long as we used Windows Forms, we could use a recursive execution of the Invoke method to use any user interface control for any read/write of its value.
In 2007, with the advent of **Windows Presentation Foundation (WPF)**, the access to user controls from asynchronous threads needed a **Dispatcher** class. From 2010, the **Task** class changed everyday programming again, as this class handled the cross-thread execution lifecycle for background tasks as efficiently as a delegate handles a call to a far method.

You understand three things:

- If a software development team chose not to use an asynchronous programming technique from the beginning, maybe relying on the DBMS speed or on an external control power, increasing data over time will do the same for latency
- On the contrary, using a time-agnostic solution will lead the team to an application that requires low maintenance over time
- If a team needs to continuously update an old application with the latest technologies available, the same winning design might lead the team to success if the technical solution changes with time

**Technical overview**

Until now, we have read about what performance requirement analysis means, how to work with performance concerns, and how to manage performance requirements against the full life cycle of a software development project. We will now learn more about the computing environment or architecture that we can leverage while programming for performance. Before getting into the details of the architecture, design, and C# specific implementations, which will be discussed in the following chapters, we will have an overview of what we could take as an advantage from each technique.

**Multithreaded programming**

Any code statement we write is executed by a processor. We can define a processor as a stupid executor of binary logic. The same processor executes a single logic every time. This is why modern operating systems work in time-sharing mode. This means the processor availability is frequently switched from virtual processors.

A thread is a virtual processor that lives within a process (the `.exe` or any .NET application) that is able to elaborate any code from any logical module of the given application.
Multicore processors are physical processors, which are all printed in the same metallic or plastic package. This helps reducing some cost and optimizing some external (but still internal to the package) devices such as memory controller, system bus, and often a high-speed cache.

Multithreading programming is the ability to program multiple threads together. This gives our applications the ability to use multiple processors, often reducing the overall execution time of our methods. Any kind of software may benefit from using multithreaded programming, such as games, server-side workflows, desktop applications, and so on. Multithreading programming is available from .NET 1.0 onward.

Although multithreading programming creates an evident performance boost by multiplying the code being executed at the same time, a disadvantage is the predictable number of threads used by the software on a system with an unpredictable number of processor cores available. For instance, by writing an application that uses two threads, we optimize the usage of a dual-core system, but we will waste the added power of a quad-core processor.

An optimization tries to split the application into the highest number of threads possible. However, although this boosts processor usage, it will also increase the overhead of designing a big hardly-coded multithreaded application.

Gaming software houses update lot of existing game engines to address multicore systems. First implementations simply used two or three main threads instead of a single one. This helped the games to use the increased available power of first multicore systems.

A simple multithreaded application, like most games made use of in 2006/2007
Parallel programming

Parallel programming adds a dynamic thread number to multithreading programming.

The thread number is then managed by the parallel framework engine itself according to internal heuristics based on dataset size, whether or not data-parallelism is used, or number of concurrent tasks, if task parallelism is used.

Parallel programming is the solution to all problems of multithreaded programming while facing a large dataset. For any other use, simply do not use parallelism, but use multithreading with a sliding elaboration design.

Parallelism is the ability to split the computation of a large dataset of items into multiple sub datasets that are to be executed in a parallel way (together) on multiple threads, with a built-in synchronization framework, the ability to unite all the divided datasets into one of the initial sizes again.

Another important advantage of parallel programming is that a parallel development framework automatically creates the right number of sub datasets based on the number CPU cores and other factors. If used on a single-core processor, nothing happens without costing any overheads to the operating system.

When a parallel computing engine splits the initial dataset into multiple smaller datasets, it creates a number, that is, a multiple of the processor core count. When the computation begins, the first group of datasets fulfils the available processor cores, while the other group waits for its time. At the end, a new dataset containing the union of all the smaller ones is created and populated with the results of all the processed dataset results.

When using parallel programming, threads flow to the cores trying to use all available resources:

Differently from hardcoded thread usage with the parallelism of Task Parallel Library items flow to all available cores
In parallel programming, the main disadvantage is the percentage of the use of parallelizable code (and data) in the overall application.

Let’s assume that we create a workflow application to read some data from an external system, process it, and then write the data back to the external system again. We can assume that if the cost of input and output is about 50 percent of the overall cost of the workflow, we can, at best, have an application that is twice as fast, if it uses all the available cores. Its the same for a 64-core CPU.

The first person to formulate this sentence was Gene Amdahl in his *Amdahl’s law* (1967). Thinking about a whole code block, we can have a speed-up that is equal to the core count only when such code presents a perfect parallelizability; otherwise, the overhead will always become a rising bottleneck as the number of cores increases. This law shows a crucial limitation of parallel programming. Not everything is parallelizable for system limitations, such as hardware resources, or because of external dependencies, such as a database that uses internal locks to grant unlimited accesses limiting parallelizability.

The following image is a preview of a 50 percent parallelizable code across a virtually infinite core count CPU:

The execution speed increase of a 50 percent un-parallelizable code. The highest speed multiplication (2X) is achieved about at 100 cores.
A software developer uses the Amdahl's law to evaluate the theoretical maximum reachable speed when using parallel computing to process a large dataset.

Against this law, another one exists, by the name of Gustafson–Barsis' law, described by John L. Gustafson and Edwin H. Barsis. They said that because of the limits software developers put on themselves, software performances do not grow in a linear way. In addition, they said that if multiple processors work on a large dataset, we can succeed processing all data in any amount of time we like; the only thing we need is enough power in the number of processor cores.

Although this is partially true only on cloud computing platform, where with the right payment plan, it is possible to have a huge availability of processor count and virtual machines. The truth is that overhead always will limit the throttling multiplication. However, this also means that we have to focus on parallelizable data and never stop trying to find a better result in our code.

**Distributed computing**

As mentioned earlier, sometimes the number of processor cores we have is never enough. Sometimes, different system categories are involved in the same software architecture. A mobile device has a fashionable body and may be very nice to use for any kind of user, while a server is powerful and can serve thousands of users, it is not mobile or nice.

Distributed computing occurs every time we split software architecture into multiple system designs. For instance, when we create a mobile application with the richest control set, multiple web services responding on multiple servers with one or more databases behind them, we create an application using distributed computing.

Here, the focus is not on speeding up a single elaboration of data, but serving multiple users. A distributed application is able to scale up and down on any virtual cloud farm or public cloud IaaS (infrastructure as a service, such as Microsoft® Azure). Although this architecture adds some possible issues, such as the security between endpoints, it also scales up at multiple nodes with the best technology its node can exploit.

The most popular distributed architecture is the n-tier; more specifically, the 3-tier architecture made by a user-interface layer (any application, including web applications), a remotely accessible business logic layer (SOAP/REST web services), and a persistence layer (one or multiple databases). As time changes, multiple nodes of any layer may be added to fulfil new demands of power. In the future, technology will add updates to a single layer to fulfill all the requirements, without forcing the other layers to do the same.
Grid computing
In grid computing, a huge dataset is divided in tiny datasets. Then, a huge number of heterogeneous systems process those small datasets and split or route them again to other small processing nodes in a huge Wide Area Network (WAN), usually the Internet itself. This is a cheaper method to achieve huge computational power with widely distributed network of commodity class systems, such as personal computers around the world.

Grid computing is definitely a customization of distributed computing, available for huge datasets of highly parallelized computational data.

In 1999, the University of California in Berkeley released the most famous project written using grid computing named SETI @ home, a huge scientific data analysis application for extra-terrestrial intelligence search. For more details, you can refer to the following link:

http://setiathome.ssl.berkeley.edu/

Summary
In this chapter, you read about the meaning and aspects of the term performance, the importance of performance engineering, and about the most widely used techniques available to fulfil any performance requirement.

In the next chapter, you will focus on the software architecture and designs that can produce well-performing applications, good-looking solutions, and avoid common mistakes.
Where to buy this book

You can buy Learning .NET High-performance Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.