

WRITING HIGH-PERFORMANCE .NET CODE



2ND
EDITION

BEN WATSON



Digitized by the Internet Archive
in 2023 with funding from
Kahle/Austin Foundation

<https://archive.org/details/writinghighperfo0000wats>

WRITING HIGH-PERFORMANCE .NET CODE

Ben Watson

Writing High-Performance .NET Code

Version 2.0

Print Edition

ISBN-13: 978-0-990-58345-5

ISBN-10: 0-990-58345-7

Copyright © 2018 Ben Watson

All Rights Reserved. These rights include reproduction, transmission, translation, and electronic storage. For the purposes of Fair Use, brief excerpts of the text are permitted for non-commercial purposes. Code samples may be reproduced on a computer for the purpose of compilation and execution and not for republication.

Trademarks

Any trademarked names, logos, or images used in this book are assumed valid trademarks of their respective owners. There is no intention to infringe on the trademark.

Disclaimer

While care has been taking to ensure the information contained in this book is accurate, the author takes no responsibility for your use of the information presented.

Contact

For more information about this book, please visit <http://www.writinghighperf.net> or email feedback@writinghighperf.net.

Cover Design

Cover design by Claire Watson, <http://www.bluekittycréations.co.uk>.

Contents

About the Author	iv
Acknowledgements	v
Foreword	vii
Introduction to the Second Edition	xiii
Introduction	xv
Purpose of this Book	xv
Why Should You Choose Managed Code?	xix
Is Managed Code Slower Than Native Code?	xxi
Are The Costs Worth the Benefits?	xxiii
Am I Giving Up Control?	xxiv
Work With the CLR, Not Against It	xxiv
Layers of Optimization	xxv
The Seductiveness of Simplicity	xxvii
.NET Performance Improvements Over Time	xxix
.NET Core	xxxiii
Sample Source Code	xxxv
Why Gears?	xxxv
1 Performance Measurement and Tools	1
Choosing What to Measure	1
Premature Optimization	4
Average vs. Percentiles	5
Benchmarking	7

CONTENTS

Useful Tools	9
Visual Studio	10
Performance Counters	18
ETW Events	26
PerfView	29
CLR Profiler	37
Windows Performance Analyzer	40
WinDbg	43
CLR MD	49
IL Analyzers	54
MeasureIt	56
BenchmarkDotNet	57
Code Instrumentation	60
SysInternals Utilities	60
Database	63
Other Tools	64
Measurement Overhead	64
Summary	65
2 Memory Management	67
Memory Allocation	68
Garbage Collection Operation	71
Detailed Heap Layout	76
Configuration Options	78
Workstation vs. Server GC	78
Background GC	80
Latency Modes	81
Large Objects	84
Advanced Options	85
Performance Tips	88
Reduce Allocation Rate	88
The Most Important Rule	89
Reduce Object Lifetime	91
Balance Allocations	92
Reduce References Between Objects	92
Avoid Pinning	93

Avoid Finalizers	94
Avoid Large Object Allocations	97
Avoid Copying Buffers	98
Pool Long-Lived and Large Objects	102
Reduce Large Object Heap Fragmentation	110
Force Full GCs in Some Circumstances	111
Compact the Large Object Heap On-Demand	112
Get Notified of Collections Before They Happen	113
Use Weak References For Caching	117
Dynamically Allocate on the Stack	126
Investigating Memory and GC	127
Performance Counters	127
ETW Events	130
What Does My Heap Look Like?	132
How Long Does a Collection Take?	136
Where Are My Allocations Occurring?	140
What Are All The Objects On The Heap?	144
Where Is the Memory Leak?	151
How Big Are My Objects?	158
Which Objects Are Being Allocated On the LOH?	162
What Objects Are Being Pinned?	164
Where Is Fragmentation Occuring?	166
What Generation Is An Object In?	173
Which Objects Survive Gen 0?	174
Who Is Calling GC.Collect Explicitly?	178
What Weak References Are In My Process?	178
What Finalizable Objects Are On The Heap?	179
Summary	180
3 JIT Compilation	183
Benefits of JIT Compilation	184
JIT in Action	185
JIT Optimizations	189
Reducing JIT and Startup Time	190
Optimizing JITting with Profiling (Multicore JIT)	192
When to Use NGEN	193

CONTENTS

Optimizing NGEN Images	195
.NET Native	196
Custom Warmup	198
When JIT Cannot Compete	199
Investigating JIT Behavior	200
Performance Counters	200
ETW Events	201
What Code Is Jitted?	202
What Methods and Modules Take the Longest To JIT?	205
Examine JITted code	206
Summary	208
4 Asynchronous Programming	209
The Thread Pool	211
The Task Parallel Library	213
Task Cancellation	217
Handling exceptions	219
Child Tasks	224
TPL Dataflow	226
A TPL Dataflow Example	228
Parallel Loops	233
Performance Tips	237
Avoid Blocking	237
Avoid Lock and Dispatch Convoys	238
Use Tasks for Non-Blocking I/O	238
<code>async</code> and <code>await</code>	244
A Note On Program Structure	247
Use Timers Correctly	249
Ensure Good Thread Pool Size	252
Do Not Abort Threads	253
Do Not Change Thread Priorities	253
Thread Synchronization and Locks	254
Do I Need to Care About Performance At All?	255
Do I Need a Lock At All?	255
Synchronization Preference Order	257
Memory Models	258

Use <code>volatile</code> When Necessary	260
Use <code>Monitor (lock)</code>	261
Use <code>Interlocked</code> Methods	264
Asynchronous Locks	268
Other Locking Mechanisms	271
Concurrency and Collections	272
Copy Your Resource Per-Thread	275
Investigating Threads and Contention	277
Performance Counters	277
ETW Events	278
Get Thread Information	279
Visualizing Tasks and Threads with Visual Studio	280
Using PerfView to find Lock Contention	282
Where Are My Threads Blocking On I/O?	282
Summary	283
5 General Coding and Class Design	285
Classes and Structs	285
A Mutable <code>struct</code> Exception: Field Hierarchies	288
Virtual Methods and Sealed Classes	289
Properties	290
Override Equals and GetHashCode for Structs	291
Thread Safety	293
Tuples	293
Interface Dispatch	294
Avoid Boxing	296
<code>ref</code> returns and locals	298
<code>for</code> vs. <code>foreach</code>	303
Casting	306
P/Invoke	308
Disable Security Checks for Trusted Code	310
Delegates	311
Exceptions	314
<code>dynamic</code>	316
Reflection	319
Code Generation	321

CONTENTS

Template Creation	322
Delegate Creation	323
Method Arguments	324
Optimization	326
Wrapping Up	327
Preprocessing	329
Investigating Performance Issues	329
Performance Counters	329
ETW Events	330
Finding Boxing Instructions	330
Discovering First-Chance Exceptions	333
Summary	335
6 Using the .NET Framework	337
Understand Every API You Call	338
Multiple APIs for the Same Thing	339
Collections	340
Collections to Avoid	340
Arrays	341
Generic Collections	345
Concurrent Collections	348
Bit-manipulation Collections	350
Initial Capacity	351
Key Comparisons	352
Sorting	353
Creating Your Own Collection Types	354
Strings	354
String Comparisons	355
ToUpper and ToLower	356
Concatenation	356
Formatting	358
ToString	359
Avoid String Parsing	360
Substrings	360
Avoid APIs that Throw Exceptions Under Normal Circumstances	361
Avoid APIs That Allocate From the Large Object Heap	361

Use Lazy Initialization	362
The Surprisingly High Cost of Enums	364
Tracking Time	366
Regular Expressions	368
LINQ	370
Reading and Writing Files	375
Optimizing HTTP Settings and Network Communication	377
SIMD	380
Investigating Performance Issues	382
Performance Counters	382
Summary	383
7 Performance Counters	385
Consuming Existing Counters	386
Creating a Custom Counter	386
Averages	387
Instantaneous	389
Deltas	389
Percentages	390
Summary	391
8 ETW Events	393
Defining Events	394
Consume Custom Events in PerfView	399
Create a Custom ETW Event Listener	401
Get Detailed EventSource Data	407
Consuming CLR and System Events	409
Custom PerfView Analysis Extension	411
Summary	414
9 Code Safety and Analysis	415
Understanding the OS, APIs, and Hardware	415
Restrict API Usage in Certain Areas of Your Code	416
Custom FxCop Rules	417
.NET Compiler Code Analyzers	425
Centralize and Abstract Performance-Sensitive and Difficult Code	436

CONTENTS

Isolate Unmanaged and Unsafe Code	436
Prefer Code Clarity to Performance Until Proven Otherwise	437
Summary	438
10 Building a Performance-Minded Team	439
Understand the Areas of Critical Performance	439
Effective Testing	440
Performance Infrastructure and Automation	442
Believe Only Numbers	444
Effective Code Reviews	445
Education	446
Summary	447
Appendices	447
A Kick-Start Your Application’s Performance	449
Define Metrics	449
Analyze CPU Usage	450
Analyze Memory Usage	450
Analyze JIT	452
Analyze Asynchronous Performance	452
B Higher-Level Performance	455
ASP.NET	455
ADO.NET	457
WPF	457
C Big O Notation	459
Big O	459
Common Algorithms and Their Complexity	463
D Bibliography	465
Useful Resources	465
People and Blogs	466
Contact Information	467

Index

469

About the Author

Ben Watson has been a software engineer at Microsoft since 2008. On the Bing platform team, he has built one of the world's leading .NET-based, high-performance server applications, handling high-volume, low-latency requests across thousands of machines for millions of customers. In his spare time, he enjoys books, music, the outdoors, and spending time with his wife Leticia and children Emma and Matthew. They live near Seattle, Washington, USA.

Acknowledgements

Thank you to my wife Leticia and our children Emma and Matthew for their patience, love, and support as I spent yet more time away from them to come up with a second edition of this book. Leticia also did significant editing and proofreading and has made the book far more consistent than it otherwise would have been.

Thank you to Claire Watson for doing the beautiful cover art for both book editions.

Thank you to my mentor Mike Magruder who has read this book perhaps more than anyone. He was the technical editor of the first edition and, for the second edition, took time out of his retirement to wade back into the details of .NET.

Thank you to my beta readers who provided invaluable insight into wording, topics, typos, areas I may have missed, and so much more: Abhinav Jain, Mike Magruder, Chad Parry, Brian Rasmussen, and Matt Warren. This book is better because of them.

Thank you to Vance Morrison who read an early version of this and wrote the wonderful Foreword to this edition.

Finally, thank you to all the readers of the first edition, who with their invaluable feedback, have also helped contribute to making the second edition a better book in every way.

Foreword

by Vance Morrison

Kids these days have no idea how good they have it! At the risk of being branded as an old curmudgeon, I must admit there is more than a kernel of truth in that statement, at least with respect to performance analysis. The most obvious example is that “back in my day” there weren’t books like this that capture both the important “guiding principles” of performance analysis as well as the practical complexities you encounter in real world examples. This book is a gold mine and is worth not just reading, but re-reading as you do performance work.

For over 10 years now, I have been the performance architect for the .NET Runtime. Simply put, my job is to make sure people who use C# and the .NET runtime are happy with the performance of their code. Part of this job is to find places inside the .NET Runtime or its libraries that are inefficient and get them fixed, but that is not the hard part. The hard part is that 90% of the time the performance of applications is not limited by things under the runtime’s control (e.g., quality of the code generation, just in time compilation, garbage collection, or class library functionality), but by things under the control of the application developer (e.g., application architecture, data structure selection, algorithm selection, and just plain old bugs). Thus my job is much more about *teaching* than *programming*.

So a good portion of my job involves giving talks and writing articles, but mostly acting as a consultant for other teams who want advice about how to make their programs faster. It is in the latter context that I first encountered Ben Watson over 6 years ago. He was “that guy on the Bing team” who always asked the non-trivial questions (and finds bugs in our code not his). Ben was

clearly a “performance guy.” It is hard to express just how truly rare that is. Probably 80% of all programmers will probably go through most of their *career* having only the vaguest understanding of the performance of the code they write. Maybe 10% care enough about performance that they learned how to use a performance tool like a profiler at all. The fact that you are reading this book (and this Foreword!) puts you well into the elite 1% that really care about performance and really want to improve it in a systematic way. Ben takes this a number of steps further: He is not only curious about anything having to do with performance, he also cares about it deeply enough that he took the time to lay it out clearly and write this book. He is part of the .0001%. You are learning from the best.

This book is important. I have seen a lot of performance problems in my day, and (as mentioned) 90% of the time the problem is in the application. This means the problem is in *your* hands to solve. As a preface to some of my talks on performance I often give this analogy: Imagine you have just written 10,000 lines of new code for some application, and you have just gotten it to compile, but you have not run it yet. What would you say is the probability that the code is bug free? Most of my audience quite rightly says zero. Anyone who has programmed knows that there is *always* a non-trivial amount of time spent running the application and fixing problems before you can have *any* confidence that the program works properly. Programming is *hard*, and we only get it right through successive refinement. Okay, now imagine that you spent some time debugging your 10,000-line program and now it (seemingly) works properly. But you also have some rather non-trivial performance goals for your application. What you would say the probability is that it has no *performance* issues? Programmers are smart, so my audience quickly understands that the likelihood is also close to zero. In the same way that there are plenty of runtime issues that the compiler can’t catch, there are plenty of performance issues that normal functional testing can’t catch. Thus everyone needs *some* amount of “performance training” and that is what this book provides.

Another sad reality about performance is that the hardest problems to fix are the ones that were “baked into” the application early in its design. That is because that is when the basic representation of the data being manipulated was chosen, and that representation places strong constraints on performance. I have lost count of the number of times people I consult with chose a poor rep-

resentation (e.g., XML, or JSON, or a database) for data that is critical to the performance of their application. They come to me for help very late in their product cycle hoping for a miracle to fix their performance problem. Of course I help them measure and we usually can find something to fix, but we can't make major gains because that would require changing the basic representation, and that is too expensive and risky to do late in the product cycle. The result is the product is never as fast as it could have been with just a small amount of performance awareness at the right time.

So how do we prevent this from happening to *our* applications? I have two simple rules for writing high-performance applications (which are, not coincidentally, a restatement of Ben's rules):

1. Have a Performance Plan
2. Measure, Measure, Measure

The “Have a Performance Plan” step really boils down to “care about perf.” This means identifying what metric you care about (typically it is some elapsed time that human beings will notice, but occasionally it is something else), and identifying the major operations that might consume too much of that metric (typically the “high volume” data operation that will become the “hot path”). Very early in the project (before you have committed to any large design decision) you should have thought about your performance goals, and *measured* something (e.g., similar apps in the past, or prototypes of your design) that *either* gives you confidence that you can reach your goals *or* makes you realize that hitting your perf goals may not be easy and that more detailed prototypes and experimentation will be necessary to find a better design. There is no rocket science here. Indeed some performance plans take literally minutes to complete. The key is that you do this early in the design so performance has a chance to influence early decisions like data representation.

The “Measure, Measure, Measure” step is really just emphasizing that this is what you will spend most of your time doing (as well as interpreting the results). As “Mad-Eye” Moody would say, we need “constant vigilance.” You can lose performance at pretty much *any* part of the product cycle from design to

FOREWORD

maintenance, and you can only prevent this by measuring again and again to make sure things stay on track. Again, there is no rocket science needed – just the will to do it on an ongoing basis (preferably by automating it).

Easy right? Well here is the rub. *In general*, programs can be complex and run on complex pieces of hardware with many abstractions (e.g., memory caches, operating systems, runtimes, garbage collectors, etc.), and so it really is not that surprising that the performance of such complex things can also be complex. There *can* be a lot of important details. There is an issue of errors, and what to do when you get conflicting or (more often) highly variable measurements. Parallelism, a great way to improve the performance of many applications also makes the analysis of that performance more complex and subject to details like CPU scheduling that previously never mattered. The subject of performance is a many-layered onion that grows ever more complex as you peel back the layers.

Taming that complexity is the value of this book. Performance can be overwhelming. There are so many things that can be measured as well as tools to measure them, and it is often not clear what measurements are valuable, and what the proper relationship among them is. This book starts you off with the basics (set goals that *you* care about), and points you in the right direction with a small set of tools and metrics that have proven their worth time and time again. With that firm foundation, it starts “peeling back the onion” to go into details on topics that become important performance considerations for some applications. Topics include things like memory management (garbage collection), “just in time” (JIT) compilation, and asynchronous programming. Thus it gives you the detail you need (runtimes are complex, and sometimes that complexity shows through and is important for performance), but in an overarching framework that allows you to connect these details with something you really care about (the goals of your application).

With that, I will leave the rest in Ben’s capable hands. The goal of my words here are not to enlighten but simply motivate you. Performance investigation is a complex area of the already complex area of computer science. It will take some time and determination to become proficient in it. I am not here to sugar-coat it, but I am here to tell you that it is worth it. Performance *does* matter. I can almost guarantee you that if your application is widely used, then its performance *will* matter. Given this importance, it is almost a crime that so few

people have the skills to systematically create high-performance applications. You are reading this now to become a member of this elite group. This book will make it *so* much easier.

Kids these days—they have no idea how good they have it!

Vance Morrison

Performance Architect for the .NET Runtime

Microsoft Corporation

Introduction to the Second Edition

The fundamentals of .NET performance have not changed much in the years since the first edition of *Writing High-Performance .NET Code*. The rules of optimizing garbage collection still remain largely the same. JIT, while improving in performance, still has the same fundamental behavior. However, there have been at least five new point releases of .NET since the previous edition, and they deserve some coverage where applicable.

Similarly, this book has undergone considerable evolution in the intervening years. In addition to new features in .NET, there were occasional and odd omissions in the first edition that have been corrected here. Nearly every section of the book saw some kind of modification, from the very trivial to significant rewrites and inclusion of new examples, material, or explanation. There are too many modifications to list every single one, but some of the major changes in this edition include:

- Overall 50% increase in content.
- Fixed all known errata.
- Incorporated feedback from hundreds of readers.
- New Foreword by .NET performance architect Vance Morrison.
- Dozens of new examples and code samples throughout.
- Revamped diagrams and graphics.

- New typesetting system for print and PDF editions.
- Added a list of CLR performance improvements over time.
- Described more analysis tools.
- Significantly increased the usage of Visual Studio for analyzing .NET performance.
- Numerous analysis examples using Microsoft.Diagnostics.Runtime (“CLR MD”).
- Added more content on benchmarking and used a popular benchmarking framework in some of the sample projects.
- New sections about CLR and .NET Framework features related to performance.
- More on garbage collection, including new information on pooling, `stackalloc`, finalization, weak references, finding memory leaks, and much more.
- Expanded discussion of different code warmup techniques.
- More information about TPL and a new section about TPL Dataflow.
- Discussion of `ref`-returns and locals.
- Significantly expanded discussion of collections, including initial capacity, sorting, and key comparisons.
- Detailed analysis of LINQ costs.
- Examples of SIMD algorithms.
- How to build automatic code analyzers and fixers.
- An appendix with high-level tips for ADO.NET, ASP.NET, and WPF.
- ... and much more!

I am confident that, even if you read the first edition, this second edition is more than worth your time and attention.

Introduction

Purpose of this Book

.NET is an amazing system for building software. It allows us to build functional, connected apps in a fraction of the time it would have taken us years ago. So much of it just works, and that is a great thing. It offers applications memory and type safety, a robust framework library, services like automatic memory management, and so much more.

Programs written with .NET are called managed applications because they depend on a runtime and framework that manages many of their vital tasks and ensures a basic safe operating environment. Unlike unmanaged, or native, software written directly to the operating system's APIs, managed applications do not have free reign of their processes.

This layer of management between your program and the computer's processor can be a source of anxiety for developers who assume that it must add some significant overhead. This book will set you at ease, demonstrate that the overhead is worth it, and that the supposed performance degradation is almost always exaggerated. Often, the performance problems developers blame on .NET are actually due to poor coding patterns and a lack of knowledge of how to optimize their programs on this framework. Skills gained from years of optimizing software written in C++, Java, or Python may not always apply to .NET managed code, and some advice is actually detrimental. Sometimes the rapid development enabled by .NET can encourage people to build bloated, slow, poorly optimized code faster than ever before. Certainly, there are other reasons why code can be of poor quality: lack of skill generally, time pressure, poor design, lack of

developer resources, laziness, and so on. This book will explicitly remove lack of knowledge about the framework as an excuse and attempt to deal with some of the others as well. With the principles explained in this book, you will learn how to build lean, fast, efficient applications that avoid these missteps. In all types of code, in all platforms, the same thing is true: if you want performant code, you have to work for it.

Performance work should never be left for the end, especially in a macro or architectural sense. The larger and more complex your application, the earlier you need to start considering performance as a major feature.

I often give the example of building a hut versus building a skyscraper. If you are building a hut, it does not really matter at what point you want to optimize some feature: Want windows? Just cut a hole in the wall. Want to add electricity? Bolt it on. You have a lot of freedom about when to completely change how things work because it is simple, with few dependencies.

A skyscraper is different. You cannot decide you want to switch to steel beams after you have built the first five floors out of wood. You must understand the requirements up front as well as the characteristics of your building materials before you start putting them together into something larger.

This book is largely about giving you an idea of the costs and benefits of your building materials, from which you can apply lessons to whatever kind of project you are building.

This is not a language reference or tutorial. It is not even a detailed discussion of the CLR. For those topics, there are other resources. (See the end of the book for a list of useful books, blogs, and people to pay attention to.) To get the most out of this book you should already have in-depth experience with .NET.

There are many code samples, especially of underlying implementation details in IL or assembly code. I caution you not to gloss over these sections. You should try to replicate my results as you work through this book so that you understand exactly what is going on.

This book will teach you how to get maximum performance out of managed code, while sacrificing none or as few of the benefits of .NET as possible. You will learn good coding techniques, specific things to avoid, and perhaps most

importantly, how to use freely available tools to easily measure your performance. This book will teach you those things with minimum fluff. This book is what you need to know, relevant and concise, with no padding of the content. Most chapters begin with general knowledge and background, followed by specific tips in a cook-book approach, and finally end with a section on step-by-step measurement and debugging for many different scenarios.

Along the way you will deep-dive into specific portions of .NET, particularly the underlying Common Language Runtime (CLR) and how it manages your memory, generates your code, handles concurrency, and more. You will see how .NET's architecture both constrains and enables your software, and how your programming choices can drastically affect the overall performance of your application. As a bonus, I will share relevant anecdotes from the last nine years of building very large, complex, high-performance .NET systems at Microsoft. You will likely notice that my bias throughout this book is for server applications, but nearly everything discussed in this book is applicable to desktop, web, and mobile applications as well. Where appropriate, I will share advice for those specific platforms.

Understanding the fundamentals will give you the "why" explanations that will allow the performance tips to make sense. You will gain a sufficient understanding of .NET and the principles of well-performing code so that when you run into circumstances not specifically covered in this book, you can apply your newfound knowledge and solve unanticipated problems.

Programming under .NET is not a completely different experience from all the programming you have ever done. You will still need your knowledge of algorithms and most standard programming constructs are pretty much the same, but we are talking about performance optimizations, and if you are coming from an unmanaged programming mindset, there are very different things you need to observe. You may not have to call `delete` explicitly any more (hurray!), but if you want to get the absolute best performance, you better believe you need to understand how the garbage collector is going to affect your application.

If high availability is your goal, then you are going to need to be concerned about JIT compilation to some degree. Do you have an extensive type system? Interface dispatch might be a concern. What about the APIs in the .NET

Framework Class Library itself? Can any of those negatively influence performance? Are some thread synchronization mechanisms better than others? Have you considered memory locality when choosing collections or algorithms?

Beyond pure coding, I will discuss techniques and processes to measure your performance over time and build a culture of performance in yourself and in your team. Good performance is not something you do once and then move on. It needs constant nourishment and care so that it does not degrade over time. Investing in a good performance infrastructure will pay massive dividends over time, allowing you to automate most of the grunt work.

The bottom line is that the amount of performance optimization you get out of your application is directly proportional to the amount of understanding you have not only of your own code, but also your understanding of the framework, the operating system, and the hardware you run on. This is true of any platform you build upon.

All of the code samples in this book are in C#, the underlying IL, or occasionally x86 or x64 assembly code, but all of the principles here apply to any .NET language. Throughout this book, I assume that you are using .NET 4.5 or higher, and some examples require newer features only available in more recent versions. I strongly encourage you to consider moving to the latest version so that you can take advantage of the latest technologies, features, bug fixes, and performance improvements.

I do not talk much about specific sub-frameworks of .NET, such as WPF, WCF, ASP.NET, Windows Forms, Entity Framework, ADO.NET, or countless others. While each of those frameworks has its own issues and performance techniques, this book is about the fundamental knowledge and techniques that you must master to develop code under all scenarios in .NET. Once you acquire these fundamentals, you can apply this knowledge to every project you work on, adding domain-specific knowledge as you gain experience. I did add a small appendix in the back, however, that can give you some initial guidance if you are trying to optimize ASP.NET, ADO.NET, or WPF applications.

Overall, I hope to show that performance engineering is just that: engineering. It is not something you get for free on any platform, not even .NET.

Why Should You Choose Managed Code?

There are many reasons to choose managed code over unmanaged code:

- Safety: The compiler and runtime can enforce type safety (objects can only be used as what they really are), boundary checking, numeric overflow detection, security guarantees, and more. There is no more heap corruption from access violations or invalid pointers.
- Automatic memory management: No more `delete` or reference counting.
- Higher level of abstraction: Higher productivity with fewer bugs.
- Advanced language features: Delegates, anonymous methods, dynamic typing, and much more.
- Huge existing code base: Framework Class Library, Entity Framework, Windows Communication Framework, Windows Presentation Foundation, Task Parallel Library, and so much more.
- Easier extensibility: With reflection capabilities, it is much easier to dynamically consume late-bound modules, such as in an extension architecture.
- Phenomenal debugging: Exceptions have a lot of information associated with them. All objects have metadata associated with them to allow thorough heap and stack analysis in a debugger, often without the need for PDBs (symbol files).

All of this is to say that you can write more code quickly, with fewer bugs. You can diagnose what bugs you do have far more easily. With all of these benefits, managed code should be your default pick.

.NET also encourages use of a standard framework. In the native world, it is very easy to have fragmented development environments with multiple frameworks in use (STL, Boost, or COM, for example) or multiple flavors of smart pointers. In .NET, many of the reasons for having such varied frameworks disappear.

While the ultimate promise of true “write once, run everywhere” code is likely always a pipe dream, it is becoming more of a reality. There are three main options for portability:

1. Portable Class Libraries allow you to target Windows Desktop, Windows Store, and other types of applications with a single class library. Not all APIs are available to all platforms, but there is enough there to save considerable effort.
2. .NET Core, which is a portable version of .NET that can run on Windows, Linux, and MacOS. It can target standard PC apps, mobile devices, data centers servers, or Internet-of-Things (IoT) devices with a flexible, minimized .NET runtime. This option is rapidly gaining popularity.
3. Using Xamarin (a set of tools and libraries), you can target Android, iOS, MacOS, and Windows platforms with a single .NET codebase.

Given the enormous benefits of managed code, consider unmanaged code to have the burden of proof, if it is even an option. Will you actually get the performance improvement you think you will? Is the generated code really the limiting factor? Can you write a quick prototype and prove it? Can you do without all of the features of .NET? In a complex native application, you may find yourself implementing some of these features yourself. You do not want to be in the awkward position of duplicating someone else’s work.

Even so, there are legitimate reasons to disqualify .NET code:

- Access to the full processor instruction set, particularly for advanced data processing applications using SIMD instructions. However, this is changing. See Chapter 6 for a discussion of SIMD programming available in .NET.
- A large existing native code base. In this case, you can consider the interface between new code and the old. If you can easily manage it with a clear API, consider making all new code managed with a simple interop layer between it and the native code. You can then transition the native code to managed code over time.

- Related to the previous point: Reliance on native libraries or APIs. For example, the latest Windows features will often be available in the C/C++-based Windows SDK before there are managed wrappers. Often, no managed wrappers exist for some functionality.
- Hardware interfacing. Some aspects of interfacing with hardware will be easier with direct memory access and other features of lower-level languages. This can include advanced graphics card capabilities for games.
- Tight control over data structures. You can control the memory layout of structures in C/C++ much more than in C#.

However, even if some of the above points apply to you, it does not mean than all of your application must be unmanaged code. You can quite easily mix the two in the same application for the best of both worlds.

Is Managed Code Slower Than Native Code?

There are many unfortunate stereotypes in this world. One of them, sadly, is that managed code cannot be fast. This is not true.

What is closer to the truth is that the .NET platform makes it very easy to write slow code if you are sloppy and uncritical.

When you build your C#, VB.NET, or other managed language code, the compiler translates the high-level language to Intermediate Language (IL) and metadata about your types. When you run the code, it is just-in-time compiled (“JITted”). That is, the first time a method is executed, the CLR will invoke the JIT compiler on your IL to convert it to assembly code (e.g., x86, x64, ARM). Most code optimization happens at this stage. There is a definite performance hit on this first run, but after that you will always get the compiled version. As we will see later, there are ways around this first-time hit when it is necessary.

The steady-state performance of your managed application is thus determined by two factors:

1. The quality of the JIT compiler
2. The amount of overhead from .NET services

The quality of generated code is generally very good, with a few exceptions, and it is getting better all the time, especially quite recently.

In fact, there are some cases where you may see a significant benefit from managed code:

- Memory allocations: There is no contention for memory allocations on the heap, unlike in native applications. Some of the saved time is transferred to garbage collection, but even this can be mostly erased depending on how you configure your application. See Chapter 2 for a thorough discussion of garbage collection behavior and configuration.
- Fragmentation: Memory fragmentation that steadily gets worse over time is a common problem in large, long-running native applications. This is less of an issue in .NET applications because the heap is less susceptible to fragmentation in the first place and when it does happen, garbage collection will compact the heap.
- JITted code: Because code is JITted as it is executed, its location in memory can be more optimal than that of native code. Related code will often be co-located and more likely to fit in a single memory page or processor cache line. This leads to fewer page faults.

The answer to the question “Is managed code slower than native code?” is an emphatic “No” in most cases. Of course, there are bound to be some areas where managed code just cannot overcome some of the safety constraints under which it operates. They are far fewer than you imagine and most applications will not benefit significantly. In most cases, the difference in performance is exaggerated. In reality, hardware and architecture will often make a bigger impact than language and platform choices.

It is much more common to run across code, managed or native, that is in reality just poorly written code; e.g., it does not manage its memory well, it uses bad patterns, it defies CPU caching strategies or is otherwise unsuitable for good performance.

Are The Costs Worth the Benefits?

As with most things, there are costs and benefits to every choice. In most cases, I have found that the benefits of managed code have outweighed the costs. In fact, with intelligent coding, you can usually avoid the worst cases of all those costs yet still gain the benefits.

The cost of the services .NET provides is not free, but it is also lower than you may expect. You do not have to reduce this cost to zero (which is impossible); just reduce it to a low enough threshold that other factors in your application's performance profile are more significant.

Feature	Benefits
JITted Code	Better memory locality, reduced memory usage
Bounds Checking	Safe memory access (fewer unfindable bugs)
Type metadata overhead	Easier debugging, rich metadata, reflection, better exception handling, easy static analysis
Garbage Collection	Fast memory allocation, no bugs with calling <code>delete</code> , safe pointer access (access violations are not possible)

All of these can add up to some significant extra gains as well:

- Higher software stability
- Less downtime
- Higher developer agility

Am I Giving Up Control?

One common objection to using managed code is that it can feel like you are giving up too much control over how your program executes. This is a particular fear of garbage collection, which occurs at what feels like random and inconvenient times. For all practical purposes, however, this is not actually true. Garbage collection is largely deterministic, and you can significantly affect how often it runs by controlling your memory allocation patterns, object scope, and GC configuration settings. What you control is different from native code, but the ability is certainly there.

Work With the CLR, Not Against It

People new to managed code often view things like the garbage collector or the JIT compiler as something they have to “deal with” or “tolerate” or “work around.” This is an unproductive way to look at it. Getting great performance out of any system requires dedicated performance work, regardless of the specific frameworks you use. For this and other reasons, do not make the mistake of viewing the GC and JIT as problems that you have to fight.

As you come to appreciate how the CLR works to manage your program’s execution, you will realize that you can make many performance improvements just by choosing to work with the CLR rather than against it. All frameworks have expectations about how they are used and .NET is no exception. Unfortunately, many of these assumptions are implicit and the API does not, nor cannot, prohibit you from making bad choices.

I dedicate a large portion of this book to explaining how the CLR works so that your own choices may more finely mesh with what it expects. This is especially true of garbage collection, for example, which has very clearly delineated guidelines for optimal performance. Choosing to ignore these guidelines is a recipe for disaster. You are far more likely to achieve success by optimizing for the framework rather than trying to force it to conform to your own notions, or worse, throwing it out altogether.

Some of the advantages of the CLR can be a double-edged sword in some sense. The ease of profiling, the extensive documentation, the rich metadata, and the ETW event instrumentation allow you to find the source of problems quickly, but this visibility also makes it easier to place blame. A native program might have all sorts of similar or worse problems with heap allocations or inefficient use of threads, but since it is not as easy to see that data, the native platform will escape blame. In both the managed and native cases, often the program itself is at fault and needs to be fixed to work better with the underlying platform. Do not mistake easy visibility of the problems for a suggestion that the entire platform is the problem.

All of this is not to say that the CLR is *never* the problem, but the default choice should always be the application, never the framework, operating system, or hardware.

Layers of Optimization

Performance optimization can mean many things, depending on which part of the software you are talking about. In the context of .NET applications, think of performance in five layers:



Figure 1. Layers of abstraction—and performance priority.

At the top, you have the design, the architecture of your system, whether it be a single application or a data center-spanning array of applications that work together. This is where all performance optimization starts because it has the

greatest potential impact to overall performance. Changing your design causes all the layers below it to change drastically, so make sure you have this right first. Only then should you move down the layers.

Then you have your actual code, the algorithms you are using to process data. This is where the rubber meets the road. Most bugs, functional or performance, are at this layer. This rule of thumb is related to a similar rule with debugging: An experienced programmer will always assume their own code is buggy rather than blaming the compiler, platform, operating system, or hardware. That definitely applies to performance optimization as well.

Below your own code is the .NET Framework—the set of classes provided by Microsoft or 3rd parties that provide standard functionality for things like strings, collections, parallelism, or even full-blown sub-frameworks like Windows Communication Framework, Windows Presentation Foundation, and more. You cannot avoid using at least some portion of the framework, but most individual parts are optional. The vast majority of the framework is implemented using managed code exactly like your own application's code. (You can even read the framework code online at <http://referencesource.microsoft.com/> or from within Visual Studio.)

Below the Framework classes lies the true workhorse of .NET, the Common Language Runtime (CLR). This is a combination of managed and unmanaged components that provide services like garbage collection, type loading, JITting, and all the other myriad implementation details of .NET.

Below that is where the code hits the metal, so to speak. Once the CLR has JITted the code, you are actually running processor assembly code. If you break into a managed process with a native debugger, you will find assembly code executing. That is all managed code is—regular machine assembly instructions executing in the context of a particularly robust framework.

To reiterate, when doing performance design or investigation, you should always start at the top layer and move down. Make sure your program's structure and algorithms make sense before digging into the details of the underlying code. Macro-optimizations are almost always more beneficial than micro-optimizations.

This book is primarily concerned with those middle layers: the .NET Framework and the CLR. These consist of the “glue” that hold your program together and are often the most invisible to programmers. However, many of the tools we discuss are applicable to all layers. At the end of the book I will briefly touch on some practical and operational things you can do to encourage performance at all layers of the system.

Note that, while all the information in this book is publicly available, it does discuss some aspects of the internal details of the CLR’s implementation. These are all subject to change.

The Seductiveness of Simplicity

C# is a beautiful language. It is familiar, owing to its C++ and Java roots. It is innovative, borrowing features from functional languages and taking inspiration from many other sources while still maintaining the C# “feel.” Through it all, it avoids the complexity of a large language like C++. It remains quite easy to get started with a limited syntax in C# and gradually increase your knowledge to use more complex features.

.NET, as a framework, is also easy to jump into. For the most part, APIs are organized into logical, hierarchical structures that make it easy to find what you are looking for. The programming model, rich libraries, and helpful IntelliSense in Visual Studio allow anyone to quickly write a useful piece of software.

However, with this ease comes a danger. As a former colleague of mine once said:

“Managed code lets mediocre developers write lots of bad code really fast.”

An example may prove illustrative. I once came upon some code that looked a bit like this:

```
Dictionary<string, object> dict =
    new Dictionary<string, object>();
...
foreach(var item in dict)
{
    if (item.Key == "MyKey")
    {
        object val = dict["MyKey"];
        ...
    }
}
```

When I first came across it, I was stunned—how could a professional developer not know how to use a dictionary? The more I thought about it, however, I started to think that perhaps this was not so obvious a situation as I originally thought. I soon came up with a theory that might explain this. The problem is the `foreach`. I believe the code originally used a `List<T>`, and what can you use to iterate over a `List<T>`? Or any enumerable collection type? `foreach`. Its simple, flexible semantics allows it to be used for nearly every collection type. At some point, I suspect, the developer realized that a dictionary structure would make more sense, perhaps in other parts of the code. They made the change, but kept the `foreach` because, after all, it still works! Except that inside the loop, you now no longer had values, but key-value pairs. Well, simple enough to fix...

You see how it is possible we could have arrived at this situation. I could certainly be giving the original developer far too much credit, and to be clear, they have little excuse in this situation—the code is clearly buggy and demonstrates a severe lack of awareness. But I believe the syntax of C# is at least a contributing factor in this case. Its very ease seduced the developer into a little less critical care.

There are many other examples where .NET and C# work together to make things a little “too easy” for the average developer: memory allocations are trivially easy to cause; many language features hide an enormous amount of code; many seemingly simple APIs have expensive implementations because of their generic, universal nature; and so on.

The point of this book is to get you beyond this point. We all begin as mediocre developers, but with good guidance, we can move beyond that phase to truly understanding the software we write.

.NET Performance Improvements Over Time

Both the CLR and the .NET Framework are in constant development to this day and there have been significant improvements to them since version 1.0 shipped in early 2002. This section documents some of the more important changes that have occurred, especially those related to performance.

1.0 (2002)

1.1 (2003)

- IPv6 Support
- Side-by-side execution
- Security improvements

2.0 (2006)

- 64-bit support (both x64 and the now-mostly-defunct IA-64)
- Nullable types
- Anonymous methods
- Iterators
- Generics and generic collection classes
- Improved UTF-8 encoding performance
- Improved `Semaphore` class

- GC
 - Reduced fragmentation from pinning
 - Reduce occurrences of `OutOfMemoryExceptions`.

3.0 (2006)

- Introduced Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF)

3.5 (2007)

- Introduced LINQ and the necessary supporting methods throughout the framework class library

3.5 SP1 (2008)

- Significant WPF performance improvements through hardware rendering, bitmap improvements, and text rendering improvements, among many others

4.0 (2010)

- Task Parallel Library
- Parallel LINQ (PLINQ)
- `dynamic` method dispatch
- Named and optional parameters
- Improved background workstation GC

4.5 (2012)

- Regular expression resolution timeout
- `async` and `await`
- GC improvements
 - Background server GC
 - Large object heap balancing for server GC
 - Better support for more than 64 processors
 - Sustained low-latency mode
 - Less LOH fragmentation
 - Datasets larger than 2 GB
- Multi-core JIT to improve startup time
- Added `WeakReference<T>`

4.5.1 (2013)

- Improved debugger support, especially for x64 code
- Automatic assembly binding redirection
- Explicit LOH compaction

4.5.2 (2014)

- ETW improvements
- Better profiling support

4.6 (2015)

- Improved 64-bit JIT (codename: RyuJIT), support for SSE2 and AVX2 instructions

INTRODUCTION

- No-GC regions added

4.6.1 (2015)

- Garbage collection performance improvements
- JIT performance improvements

4.6.2 (2016)

- Allow path names longer than 260 characters
- JIT performance and reliability improvements
- Significant `EventSource` bug fixes
- GC
 - Ability to collect all objects that are next to pinned objects
 - More efficient gen 2 free space usage

4.7 (2017)

- JIT performance improvements
- Advanced GC configuration options
- `ValueTuple` type

4.7.1 (2017)

- GC improvements to LOH allocation speed. LOH allocations are no longer blocked by entire background GC sweep.

4.7.2 (2018)

- HashSet<T> and ConcurrentDictionary< TKey , TValue > performance improvements
- ReaderWriterLockSlim and ManualResetEventSlim performance improvements
- GC performance improvements

.NET Core

.NET Core is a cross-platform, open source, modular version of .NET. Microsoft released version 1.0 in June of 2016, and 2.0 was released in August of 2017. You can consider .NET Core to be a subset of the full .NET Framework, but it also contains additional APIs not available in the standard edition. With .NET Core, you can write apps for the command line, Universal Windows Platform apps, ASP.NET Core web apps, and portable code libraries. While much of the standard .NET Framework Class Library has been ported to .NET Core, there are many APIs that are not present. If you wish to migrate from .NET Framework to .NET Core, you may need to do some significant refactoring. It notably does not support Windows Forms or WPF applications.

The underlying code for both the JIT and the Garbage Collector are the same as in the full .NET Framework. The CLR functions the same in both systems.

Nearly all the performance issues discussed in this book apply equally to both systems and I will make no distinction between the two platforms.

That said, there are some important caveats:

- ASP.NET Core is a significant improvement over ASP.NET using the .NET Framework. If you want high-performance web serving, it is worth it to adopt ASP.NET Core.

- Because .NET Core is open source, it receives improvements much faster than the .NET Framework. Some of these changes are ported back to the .NET Framework, but it is not a guarantee.
- Many individual APIs have received some performance optimization work:
 - Collections such as `List<T>`, `SortedSet<T>`, `Queue<T>`, and others were improved or rewritten completely in some cases.
 - LINQ has reduced allocations and instruction count.
 - Regular expression and string processing is faster.
 - Math operations on non-primitives are faster.
 - String encoding is more efficient.
 - Network APIs are faster.
 - Concurrency primitives have been subtly improved to be faster.
 - And much more...

There are many specific technologies that do not work with .NET Core, however:

- WPF applications
- Windows Forms applications
- ASP.NET Web Forms
- WCF servers
- C++/CLI (.NET Core does support P/Invoke, however)

.NET Core is where a lot of the focus and love is. All new development should use it, when possible. Because it is open source, you yourself can contribute changes to make it even better.

Sample Source Code

This book makes frequent references to some sample projects. These are all quite small, encapsulated projects meant to demonstrate a particular principle. As simple examples, they will not adequately represent the scope or scale of performance issues you will discover in your own investigations. Consider them a starting point of techniques or investigation skills, rather than as serious examples of representative code.

You can download all of the sample code from the book's web site at <http://www.writinghighperf.net>. Most projects will build fine in .NET 4.5, but some will require 4.7. You should have at least Visual Studio 2015 to open most of the projects.

Some of the sample projects, tools, and examples in this book use NuGet packages. They should automatically be restored by Visual Studio, but you can individually manage them by right clicking on a project and selecting "Manage NuGet References."

Why Gears?

Finally, I would like to say a brief note about the cover. The image of gears has been in my mind since well before I decided to write this book. I often think of effective performance in terms of clockwork, rather than pure speed, though that is an important aspect too. You must not only write your program to do its own job efficiently, but it has to mesh well with .NET, its own internal parts, the operating system, and the hardware. Often, the right approach is just to make sure your application is not doing anything that interferes with the gearworks of the whole system, but encourages it to keep running smoothly, with minimal interruptions. This is clearly the case with things like garbage collection and asynchronous thread patterns, but this metaphor also extends to things like JIT, logging, and much more.

As you read this book, keep this metaphor in mind to guide your understanding of the various topics.

Chapter 1

Performance Measurement and Tools

Before we dive into the specifics of the CLR and .NET, we need to understand performance measurement in general, as well as the many tools available to us. You are only as powerful as the tools in your arsenal, and this chapter attempts to give you a solid grounding and set the stage for many of the tools that will be discussed throughout the book.

Choosing What to Measure

Before deciding what to measure, you need to determine a set of performance requirements. The requirements should be general enough to not prescribe a specific implementation, but specific enough to be measurable. They need to be grounded in reality, even if you do not know how to achieve them yet. These requirements will, in turn, drive which metrics you need to collect. Before collecting numbers, you need to know what you intend to measure. This sounds obvious, but it is actually a lot more involved than you may think. Consider memory. You obviously want to measure memory usage and minimize it. But which kind of memory? Private working set? Commit size? Paged pool? Peak working set? .NET heap size? Large object heap size? Individual processor heaps to

ensure they are balanced? Some other variant? For tracking memory usage over time, do you want the average for an hour, the peak? Does memory usage correlate with processing load size? As you can see, there are easily a dozen or more metrics just for the concept of memory alone. And we have not even touched the concept of private heaps or profiling the application to see what kinds of objects are using memory!

Be as specific as possible when describing what you want to measure.



In one large server application I was responsible for, we tracked its private bytes (see the section on Performance Counters in this chapter for more information about various types of memory measurement) as a critical metric and used this number to decide when we needed to do things like restart the process before beginning a large, memory-intensive operation. It turned out that quite a large amount of those “private bytes” were actually paged out over time and not contributing to the memory load on the system, which is what we were really concerned with. We changed our system to measure the working set instead. This had the benefit of “reducing” our memory usage by a few gigabytes. (As I said, this was a rather large application.)

Once you have decided what you are going to measure, come up with specific goals for each of those metrics. Early in development, these goals may be quite malleable, even unrealistic, but should still be based on the top-level requirements. The point at the beginning is not necessarily to meet the goals, but to force you to build a system that automatically measures you against those goals.

Your goals should be quantifiable. A high-level goal for your program might state that it should be “fast.” Of course it should. That is not a very good metric because “fast” is subjective and there is no well-defined way to know you are meeting that goal. You must be able to assign a number to this goal and be able to measure it.

Bad: “The user interface should be responsive.”

Good: “No operation may block the UI thread for more than 20 milliseconds.”

However, just being quantifiable is not good enough either. You need to be very specific, as we saw in the memory example earlier.

Bad: “Memory should be less than 1 GB.”

Good: “Working set memory usage should never exceed 1 GB during peak load of 100 queries per second.”

The second version of that goal gives a very specific circumstance that determines whether you are meeting your goal. In fact, it suggests a good test case.

Another major determining factor in what your goals should be is the kind of application you are writing. A user interface program must at all costs remain responsive on the UI thread, whatever else it does. A server program handling dozens, hundreds, or even thousands of requests per second must be incredibly efficient in handling I/O and synchronization to ensure maximum throughput and keep the CPU utilization high. You design a server of this type in a completely different way than other programs. It is very difficult to fix a poorly written application retroactively if it has a fundamentally flawed architecture from an efficiency perspective.

Capacity planning is also important. A useful exercise while designing your system and planning performance measurement is to consider what the optimal theoretical performance of your system is. If you could eliminate all overhead like garbage collection, JIT, thread interrupts, or whatever you deem is overhead in your application, then what is left to process the actual work? What are the theoretical limits that you can think of, in terms of workload, memory usage, CPU usage, and internal synchronization? This often depends on the hardware and OS you are running on. For example, if you have a 16-processor server with 64 GB of RAM with two 10 GB network links, then you have an idea of your parallelism threshold, how much data you can store in memory, and how much you can push over the wire every second. It will help you plan how many machines of this type you will need if one is not enough.

Premature Optimization

You have likely heard the phrase, coined by Donald Knuth, “Premature optimization is the root of all evil.” The context of the quote is in determining which areas of your program are actually important to optimize. This brings us to Amdahl’s Law, which describes the theoretical maximum speedup of a software program through optimization, in particular how it applies to sequential programs and picking which parts of a program to optimize. Micro-optimizing code that does not significantly contribute to overall inefficiency is largely a waste of time. This concept most obviously applies to micro-optimizations at the code level, but it can apply to higher levels of your design as well. You still need to understand your architecture and its constraints as you design or you will miss something crucial and severely hamstring your application. But within those parameters, there are many areas which are not important (or you do not know which sub-areas are important yet). It is not impossible to redesign an existing application from the ground up, but it is far more expensive than doing it right in the first place. When architecting a large system, often the only way you can avoid the premature optimization trap is with experience and examining the architecture of similar or representative systems. In any case, you must bake performance goals into the design up front. Performance, like security and many other aspects of software design, cannot be an afterthought, but needs to be included as an explicit goal from the start.

The performance analysis you will do at the beginning of a project is different from that which occurs once it has been written and is being tested. At the beginning, you must make sure the design is scalable, that the technology can theoretically handle what you want to do, and that you are not making huge architectural blunders that will forever haunt you. Once a project reaches testing, deployment, and maintenance phases, you will instead spend more time on micro-optimizations, analyzing specific code patterns, trying to reduce memory usage, etc.

You will never have time to optimize everything, so start intelligently. Optimize the most inefficient portions of a program first to get the largest benefit. This is why having goals and an excellent measurement system in place is critical otherwise, you do not even know where to start.

Average vs. Percentiles

When considering the numbers you are measuring, decide what the most appropriate statistics are. Most people default to average, which is certainly important in most circumstances, but you should also consider percentiles. If you have availability requirements, you will almost certainly need to have goals stated in terms of percentiles. For example:

“Average latency for database requests must be less than 10ms. The 95th percentile latency for database requests must be less than 100ms.”

If you are not familiar with this concept, it is actually quite simple. If you take 100 measurements of something and sort them, then the 95th entry in that list is the 95th percentile value of that data set. The 95th percentile says, “95% of all samples have this value or less.” Alternatively, “5% of requests have a value higher than this.”

The general formula for calculating the index of the P^{th} percentile of a sorted list is:

$$\frac{P}{100}N$$

where P is the percentile and N is the length of the list.

Consider a series of measurements for generation 0 garbage collection pause time in milliseconds with these values (pre-sorted for convenience):

1, 2, 2, 4, 5, 5, 8, 10, 10, 11, 11, 11, 15, 23, 24, 25, 50, 87

For these 18 samples, we have an average of 17ms, but the 95th percentile is much higher at 50ms. If you just saw the average number, you may not be concerned with your GC latencies, but knowing the percentiles, you have a better idea of the full picture and that there are some occasional GC's happening that are far worse.

This series also demonstrates that the median value (50th percentile) can be quite different from the average. The average value of a series of measurements is often prone to strong influence by values in the higher percentiles.

Percentiles values are usually far more important for high-availability services. The higher availability you require, the higher percentile you will want to track. Usually, the 99th percentile is as high as you need to care about, but if you deal in a truly enormous volume of requests, 99.99th, 99.999th, or even higher percentiles will be important. Often, the value you need to be concerned about is determined by business needs, not technical reasons.

Percentiles are valuable because they give you an idea of how your metrics degrade across your entire execution context. Even if the average user or request experience in your application is good, perhaps the 90th percentile metric shows some room for improvement. That is telling you that 10% of your execution is being impacted more negatively than the rest. Tracking multiple percentiles will tell you how fast this degradation occurs. How important this percentage of users or requests is must ultimately be a business decision, and there is definitely a law of diminishing returns at play here. Getting that last 1% may be extremely difficult and costly.

I stated that the 95th percentile for the above data set was 50ms. While technically true, it is not useful information in this case – there is not actually enough data to make that call with any statistical significance, and it could be just a fluke. To determine how many samples you need, just use a rule of thumb: You need one “order of magnitude” more samples than the target percentile. For percentiles from 0–99, you need 100 samples minimum. You need 1,000 samples for 99.9th percentile, 10,000 samples for 99.99th percentile, and so on. This mostly works, but if you are interested in determining the actual number of samples you need from a mathematical perspective, research sample size determination.

Put more exactly, the potential error varies with the square root of the number of samples. For example, 100 samples yields an error range of 90–100, or a 10% error; 1,000 samples yields an error range of 969–1031, or a 3% error.

Do not forget to also consider other types of statistical values: minimum, maximum, median, standard deviations, and more, depending on the type of metric you are measuring. For example, to determine statistically relevant differences between two sets of data, *t*-tests are often used. Standard deviations are used to determine how much variation exists within a data set.

Benchmarking

If you want to measure the performance of a piece of code, especially to compare it to an alternative implementation, what you want is a benchmark. The literal definition of a benchmark is a standard against which measurements can be compared. In terms of software development, this means precise timings, usually averaged across many thousands (or millions) of iterations.

You can benchmark many types of things at different levels—entire programs to single methods. However, the more variability that exists in the code under test, the more iterations you will need to achieve sufficient accuracy.

Running benchmarks is a tricky endeavor. You want to measure the code in real-world conditions to get real-world, actionable data, but creating these conditions while getting useful data can be trickier than it seems.

Benchmarks shine when they test a single, uncontended resource, the classic example being CPU time. You certainly can test things like network access time, or reading files off an SSD, but you will need to take more care to isolate those resources from outside influence. Modern operating systems are not designed for this kind of isolation, but with careful control of the environment, you can likely achieve satisfactory results.

Testing entire programs or submodules are more likely to involve this use of contended resources. Thankfully, such large-scope tests are rarely called for. A quick profile of an app will reveal those spots that use the most resources, allowing for narrow focus on those areas.

Small-scope micro-benchmarking most commonly measures the CPU time of single methods, often rerunning them millions of times to get precise statistics on the time taken.

In addition to hardware isolation, there are a number of other factors to consider:

- Code must be JITted: The first time you run a method takes a lot longer than subsequent iterations.
- Other Hidden Initialization: There are OS caches, file system caches, CLR caches, hardware caches, code generation, and myriad other startup costs that can impact the performance of code.
- Isolation: If other expensive processes are running, they can interfere with the measurements.
- Outliers: Statistical outliers in measurement must be accounted for and probably discarded. Determining what are outliers and what is normal variance can be tricky.
- Narrowly Focused: CPU time is important, but so is memory allocation, I/O, thread blocking, and more.
- Release vs. Debug Code: Benchmarking should always be done on Release code, with all optimizations turned on.
- Observer Effects: The mere act of observing something necessarily changes what is being observed. For example, measuring CPU or memory allocations in .NET involves emitting and measuring extra ETW events, something not normally done.

The sample code that accompanies this book has a few quick-and-dirty benchmarks throughout, but for the above reasons, they should not be taken as the absolute truth.

Instead of writing your own benchmarks, you should almost certainly use an existing library that handles many of the above issues for you. I'll discuss a couple of options later in this chapter.

Useful Tools

If there is one single rule that is the most important in this entire book, it is this:

Measure, Measure, Measure!

You do NOT know where your performance problems are if you have not measured accurately. You will definitely gain experience and that can give you some strong hints about where performance problems are, just from code inspection or gut feel. You may even be right, but resist the urge to skip the measurement for anything but the most trivial of problems. The reasons for this are two-fold:

First, suppose you are right, and you have accurately found a performance problem. You probably want to know how much you improved the program, right? Bragging rights are much more secure with hard data to back them up.

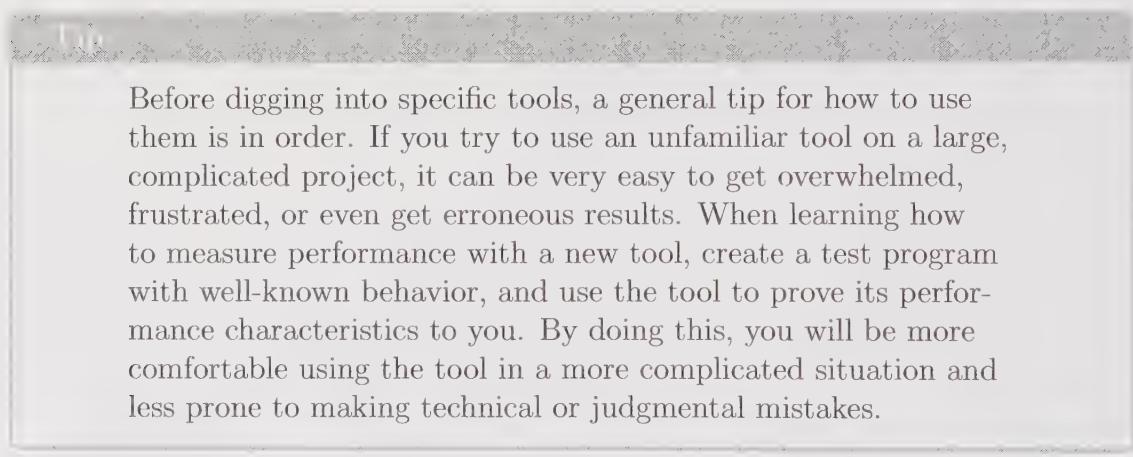
Second, I cannot tell you how often I have been wrong. Case in point: While analyzing the amount of native memory in a process compared to managed memory, we assumed for a while that it was coming from one particular area that loaded an enormous data set. Rather than putting a developer on the task of reducing that memory usage, we did some experiments to disable loading that component. We also used the debugger to dump information about all the heaps in the process. To our surprise, most of the mystery memory was coming from assembly loading overhead, not this dataset. We saved a lot of wasted effort.

Optimizing performance is meaningless if you do not have effective tools for measuring it. Performance measurement is a continual process that you should bake into your development tool set, testing processes, and monitoring tools. If your application requires continual monitoring for functionality purposes, then it likely also requires performance monitoring.

The remainder of this chapter covers various tools that you can use to profile, monitor, and debug performance issues. I give emphasis to Visual Studio and software that is freely available, but know there are many other commercial offerings that can in some cases simplify various analysis tasks. If you have the

budget for these tools, go for it. However, there is a lot of value in using some of the leaner tools I describe (or others like them). For one, they may be easier to run on customer machines or production environments. More importantly, by being a little “closer to the metal,” they will encourage you to gain knowledge and understanding at a very deep level that will help you interpret data, regardless of the tool you are using.

For each of the tools, I describe basic usage and general knowledge to get started. Sections throughout the book will give you detailed steps for very specific scenarios, but will often rely on you already being familiar with the UI and the basics of operation.



Before digging into specific tools, a general tip for how to use them is in order. If you try to use an unfamiliar tool on a large, complicated project, it can be very easy to get overwhelmed, frustrated, or even get erroneous results. When learning how to measure performance with a new tool, create a test program with well-known behavior, and use the tool to prove its performance characteristics to you. By doing this, you will be more comfortable using the tool in a more complicated situation and less prone to making technical or judgmental mistakes.

Visual Studio

While it is not the only IDE, most .NET programmers use Visual Studio, and if you do, chances are this is where you will start to analyze performance. Different versions of Visual Studio come with different tools. This book will assume you have at least the Professional version installed, but I will also describe some tools found in higher versions as well. If you do not have the right version, then skip ahead to the other tools mentioned.

Assuming you installed Visual Studio Professional or higher, you can access the performance tools via the Analyze menu and selecting Performance Profiler (or use the default keyboard shortcut: Alt+F2).

Standard .NET applications will show at least three options, with more available depending on the specific type of application:

- CPU Usage: Measures CPU usage per function.
- Memory Usage: Shows garbage collections and allows you to take heap snapshots.
- Performance Wizard: Uses VsPerf.exe to do ETW-based analysis of CPU usage (sampling or instrumentation), .NET memory allocation, and thread contention.

Analysis Target



Available Tools

<input type="checkbox"/> CPU Usage See where the CPU is spending time executing your code. Useful when the CPU is the performance bottleneck	<input type="checkbox"/> Memory Usage Investigate application memory to find issues such as memory leaks
<input type="checkbox"/> Performance Wizard CPU Sampling, Instrumentation, .NET Memory allocation, and Resource Contention	

Figure 1.1. Profiling options in Visual Studio.

If you just need to analyze CPU or look at what is on the heap, then use the first two tools. The Performance Wizard can also do CPU analysis, but it can be a bit slower. However, despite being somewhat of a legacy tool, it can also track memory allocations and concurrency.

For superior concurrency analysis, install the free Concurrency Visualizer, available as an optional extension (Tools | Extensions and Updates... menu).

The Visual Studio tools are among the easiest to use, but if you do not already have the right version of Visual Studio, they are quite expensive. They are also fairly limited and inflexible in what they provide. If you cannot use Visual Studio, or need more capabilities, I describe free alternatives below. Nearly all modern performance measurement tools use the same underlying mechanism (at least in Windows 8/Server 2012 and above kernels): ETW events. ETW stands for Event Tracing for Windows and this is the operating system's way of logging all interesting events in an extremely fast, efficient manner. Any application can generate these events with simple APIs. Chapter 8 describes how to take advantage of ETW events in your own programs, defining your own or integrating with a stream of system events. Some tools, such as PerfView, can collect arbitrary ETW events all at once and you can analyze all of them separately from one collection session. Sometimes I think of Visual Studio performance analysis as “development-time” while the other tools are for the real system. Your experience may differ and you should use the tools that give you the most bang for the buck.

CPU Profiling

This section will introduce the general interface for profiling with the CPU profiling options. The other profiler options (such as for memory) will be covered later in the book, in appropriate sections.

When you choose CPU Usage, the results will bring up a window with a graph of CPU usage and a list of expensive methods.

If you want to drill into a specific method, just double-click it on the list, and it will open up a method Call/Callee view.

If that option does not give you enough information, take a look at the performance wizard. This tool uses VsPerf.exe to gather important events.

When you choose the CPU (Sampling), it collects CPU samples without any interruption to your program.

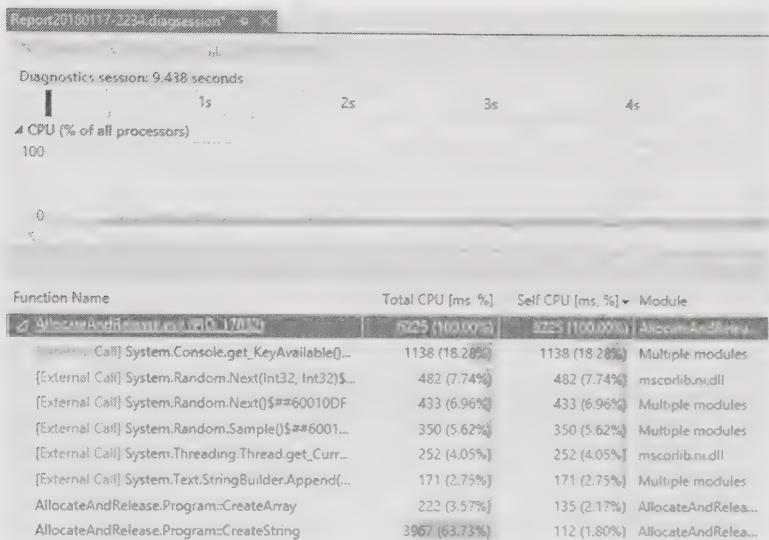


Figure 1.2. CPU Usage results. Timeline, overall usage graph, and tree of the most expensive methods.

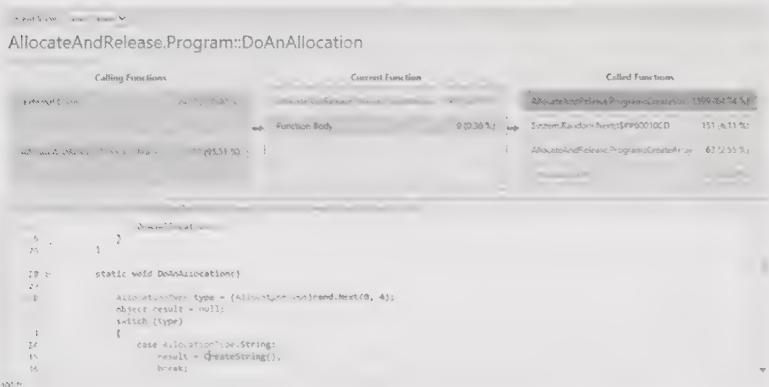


Figure 1.3. CPU Usage Method Call/Callee Diagram. Shows the most expensive parts of a method.

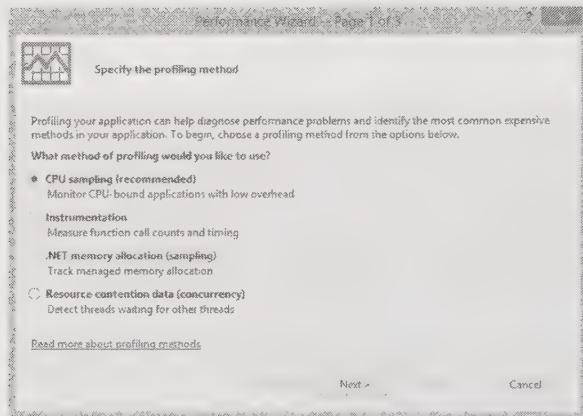


Figure 1.4. The first screen of the Performance Wizard.

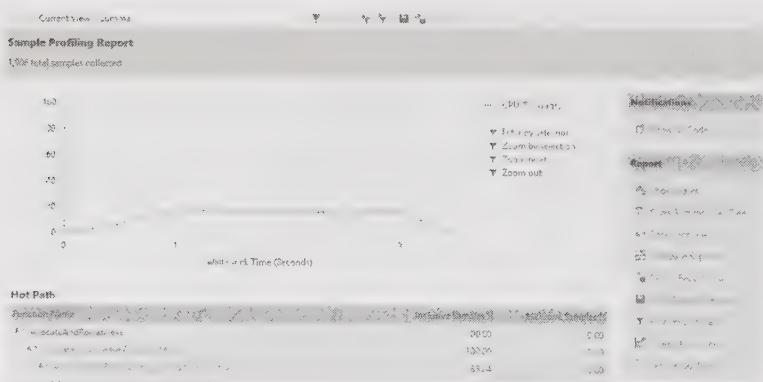


Figure 1.5. The Performance Wizard's CPU sampling report view.

While a different interface than the CPU Usage view we saw earlier, this view shows you the overall CPU usage on a time line, with a tree of expensive methods below it. There are also alternate reports you can view. You can zoom in on the graph and the rest of the analysis will update in response. Clicking on a method name in the table will take you to a familiar-looking Function Details view.

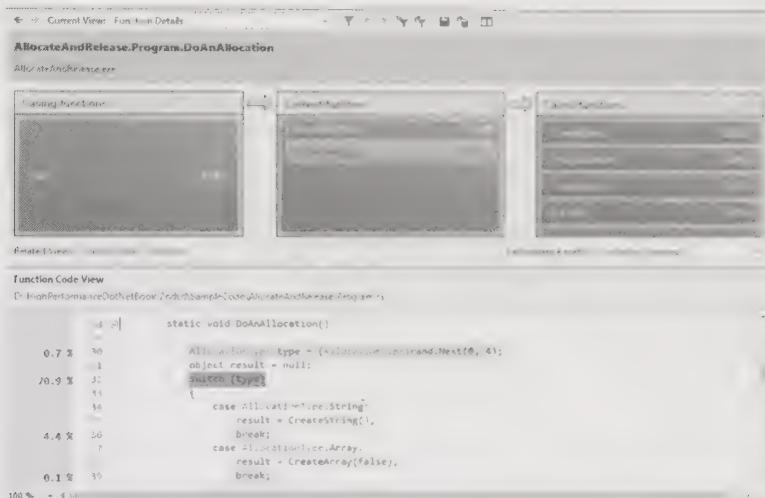


Figure 1.6. Details of the method's CPU usage.

Below the function call summary, you will see the source code (if available), with highlighted lines showing the most expensive parts of the method.

There are other reports as well, including:

- Modules: Which assemblies have the most samples in them.
- Caller/Callee: An alternative to the Function Details view that shows tables of samples above and below the current method in the stack.
- Functions: A quick way to see a table of all functions in the process.
- Lines: A way to jump quickly to the most expensive individual code lines in the process.

Instead of sampling, you can choose to instrument the code. This modifies the original executable by adding instructions around each method call to measure the time spent. This can give more accurate reporting for very small, fast methods, but it has much higher overhead in execution time as well as the amount of data produced. Other than a lack of a CPU graph, the report looks and behaves the same as the CPU sampling report. The major difference in the interface is that it is measuring time instead of number of samples.

Command Line Profiling

Visual Studio can analyze CPU usage, memory allocations, and resource contentions. This is perfect for use during development or when running comprehensive tests that accurately exercise the product. However, it is very rare for a test to accurately capture the performance characteristics of a large application running on real data. If you need to capture performance data on non-development machines, say a customer's machine or in the data center, you need a tool that can run outside of Visual Studio.

For that, there is the Visual Studio Standalone Profiler, which comes with the Professional or higher versions of Visual Studio. You will need to install it from your installation media separately from Visual Studio. On my ISO images for both 2012 - 2015 Professional versions, it is in the Standalone Profiler directory. For Visual Studio 2017, the executable is VsPerf.exe and is located in `C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\Team Tools\Performance Tools`.

To collect data from the command line with this tool:

1. Navigate to the installation folder (or add the folder to your path)
2. Run: `VsPerfCmd.exe /Start:Sample /Output:outputfile.vsp`
3. Run the program you want to profile
4. Run: `VsPerfCmd.exe /Shutdown`

This will produce a file called `outputfile.vsp`, which you can open in Visual Studio.

`VsPerfCmd.exe` has a number of other options, including all of the profiling types that the full Visual Studio experience offers. Aside from the most common option of Sample, you can choose:

- Coverage: Collects code coverage data
- Concurrency: Collects resource contention data
- Trace: Instruments the code to collect method call timing and counts

Trace vs. Sample mode is an important choice. Which one to use depends on what you want to measure. Sample mode should be your default. It interrupts the process every few milliseconds and records the stacks of all threads. This is the best way to get a good picture of CPU usage in your process. However, it does not work well for I/O calls, which will not have much CPU usage, but may still contribute to your overall run time.

Trace mode requires modification of every function call in the process to record time stamps. It is much more intrusive and causes your program to run much slower. However, it records actual time spent in each method, so may be more accurate for smaller, faster methods.

Coverage mode is not for performance analysis, but is useful for seeing which lines of your code were executed. This is a nice feature to have when running tests to see how much of your product the tests cover. There are commercial products that do this for you, but you can do it yourself without much more work.

Concurrency mode records events that occur when there is contention for a resource via a lock or some other synchronization object. This mode can tell you if your threads are being blocked due to contention. See Chapter 4 for more information about asynchronous programming and measuring the amount of lock contention in your application.

Performance Counters

Performance counters are some of the simplest ways to monitor your application's and the system's performance. Windows has hundreds of counters in dozens of categories, including many for .NET. The easiest way to access these is via the built-in Windows utility PerformanceMonitor (PerfMon.exe).

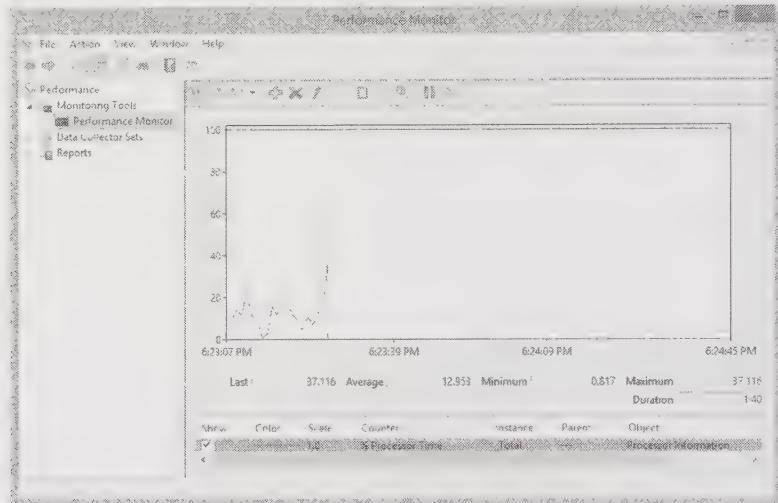


Figure 1.7. PerfMon's main window showing a processor counter for a small window of time. The vertical line represents the current instance and the graph will wrap around after 100 seconds by default.

Each counter has a category and a name. Many counters also have instances of the selected counter as well. For example, for the % Processor Time counter in the Process category, the instances are the various processes for which there are values. Some counters also have meta-instances, such as Total or <Global>, which aggregate the values over all instances.

Many of the chapters ahead will detail the relevant counters for that topic, but there are general-purpose counters that are not .NET-specific that you should be familiar with. There are performance counters for nearly every Windows subsystem and these are generally applicable to every program.

However, before continuing, you should familiarize yourself with some basic operating system terminology:

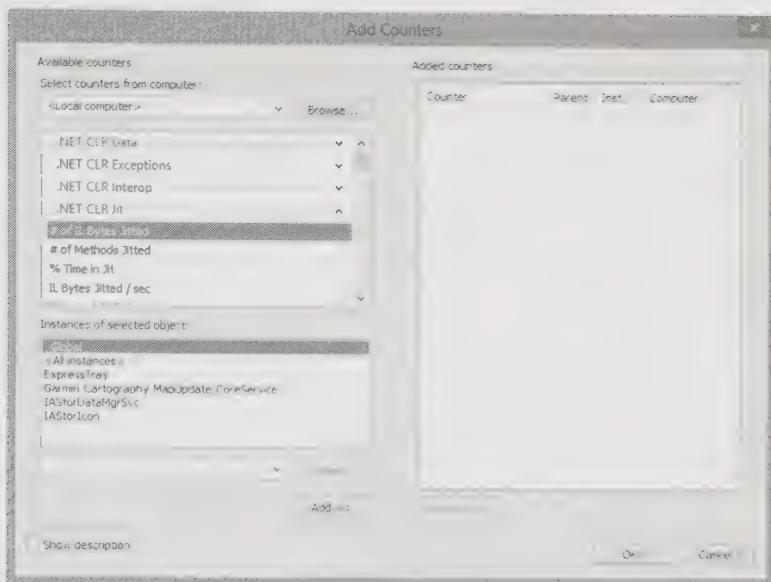


Figure 1.8. One of the hundreds of counters in many categories, showing all of the applicable instances (processes, in this case).

- **Physical Memory:** The actual physical memory chips in a computer. Only the operating system manages physical memory directly.
- **Virtual Memory:** A logical organization of memory in a given process. Virtual memory size can be larger than physical memory. For example, 32-bit programs have a 4 GB address space, even if the computer itself only has 2 GB of RAM. Windows allows the program to access only 2 GB of that by default, but all 4 GB is possible if the executable has the large-address aware bit set. (On 32-bit versions of Windows, large-address aware programs are limited to 3 GB, with 1 GB reserved for the operating system.) As of Windows 8.1 and Server 2012, 64-bit processes have a 128 TB address space, far larger than the 4 TB physical memory limit. Some of the virtual memory may be in RAM while other parts are stored on disk in a paging file. Contiguous blocks of virtual memory may not be contiguous in physical memory. All memory addresses in a process are for the virtual memory.

- **Reserved Memory:** A region of virtual memory address space that has been reserved for the process and thus will not be allocated to a future requester. Reserved memory cannot be used for memory allocation requests because there is nothing backing it – it is just a description of a range of memory addresses.
- **Committed Memory:** A region of memory that has a physical backing store. This can be RAM or disk.
- **Page:** An organizational unit of memory. Blocks of memory are allocated in a page, which is usually a few KB in size.
- **Paging:** The process of transferring pages between regions of virtual memory. The page can move to or from another process (soft paging) or the disk (hard paging). Soft paging can be accomplished very quickly by mapping the existing memory into the current process's virtual address space. Hard paging involves a relatively slow transfer of data to or from a disk. Your program must avoid this at all costs to maintain good performance.
- **Page In:** Transfer a page from another location to the current process.
- **Page Out:** Transfer a page from the current process to another location, such as disk.
- **Context Switch:** The process of saving and restoring the state of a thread or process. Because there are usually more running threads than available processors, there are often many context switches per second. These are pure overhead, so fewer is better, but it is difficult to know what an optimal absolute value should be.
- **Kernel Mode:** A mode that allows the OS to modify low-level aspects of the hardware's state, such as modifying certain registers or enabling/disabling interrupts. Transitioning to Kernel Mode requires an operating system call, and can be quite expensive.
- **User Mode:** An unprivileged mode of executing instructions. There is no ability to modify low-level aspects of the system.

I will use some of these terms throughout the book, especially in Chapter 2 when I discuss garbage collection. For more information on these topics, look at a dedicated operating system book such as *Windows Internals*. (See the bibliography at the end of the book.)

The Process category of counters surfaces much of this critical information via counters with instances for each process, including:

- **% Privileged Time:** Amount of time spent in executing privileged (kernel mode) code.
- **% Processor Time:** Percentage of a single processor the application is using. If your application is using two logical processor cores at 100% each, then this counter will read 200.
- **% User Time:** Amount of time spent in executing unprivileged (user mode) code.
- **IO Data Bytes/sec:** How much I/O your process is doing.
- **Page Faults/sec:** Total number of page faults in your process. A page fault occurs when a page of memory is missing from the current working set. It is important to realize that this number includes both soft and hard page faults. Soft page faults are innocuous and can be caused by the page being in memory, but outside the current process (such as for shared DLLs). Hard page faults are more serious, indicating data that is on disk but not currently in memory. Unfortunately, you cannot track hard page faults per process with performance counters, but you can see it for the entire system with the Memory\Page Reads/sec counter. You can do some correlation with a process's total page faults plus the system's overall page reads (hard faults). You can definitively track a process's hard faults with ETW tracing with the Windows Kernel/Memory/Hard Fault event.
- **Pool Nonpaged Bytes:** Typically operating system and driver allocated memory for data structures that cannot be paged out such as operating system objects like threads and mutexes, but also custom data structures.
- **Pool Paged Bytes:** Also for operating system data structures, but these are allowed to be paged out.

- **Private Bytes:** Committed virtual memory private to the specific process (not shared with any other processes).
- **Virtual Bytes:** Allocated memory in the process's address space, some of which may be backed by the page file, possibly shared with other processes or private to the process.
- **Working Set:** The amount of virtual memory currently resident in physical memory (usually RAM).
- **Working Set-Private:** The amount of private bytes currently resident in physical memory.
- **Thread Count:** The number of threads in the process. This may or may not be equal to the number of .NET threads. See Chapter 4 for a discussion of .NET thread-related counters.

There are a few other generally useful categories, depending on your application. You can use PerfMon to explore the specific counters found in these categories.

- **IPv4/IPv6:** Internet Protocol-related counters for datagrams and fragments.
- **Memory:** System-wide memory counters such as overall paging, available bytes, committed bytes, and much more.
- **Objects:** Data about kernel-owned objects such as events, mutexes, processes, threads, semaphores, and sections.
- **Processor:** Counters for each logical processor in the system.
- **System:** Context switches, alignment fixes, file operations, process count, threads, and more.
- **TCPv4/TCPv6:** Data for TCP connections and segment transfers.

It is surprisingly difficult to find detailed information on performance counters on the Internet, but thankfully, they are self-documenting! In the Add Counter dialog box in PerfMon, you can check the “Show description” box at the bottom to display details on the highlighted counter.

PerfMon also has the ability to collect specified performance counters at scheduled times and store them in logs for later viewing, or even perform a custom action when a performance counter passes a threshold. You do this with Data Collector Sets and they are not limited just to performance counter data, but can also collect system configuration data and ETW events.

To set up a Data Collector Set, in the main PerfMon window:

1. Expand the Data Collector Sets tree.
2. Right-click on User Defined.
3. Select New.
4. Select Data Collector Set.
5. Give it a name, check Create manually (Advanced), and click the Next button.
6. Check the Performance counter box under Create Data Logs and click the Next button.
7. Click Add to select the counters you want to include.
8. Click Next to set the path where you want to store the logs and Next again to select security information.

Once done, you can open the properties for the collection set and set a schedule for collection. You can also run them manually by right-clicking on the job node and selecting Start. This will create a report, which you can view by double-clicking its node under Reports in the main tree view.

To create an alert, follow the same process but select the Performance Counter Alert option in the Wizard.

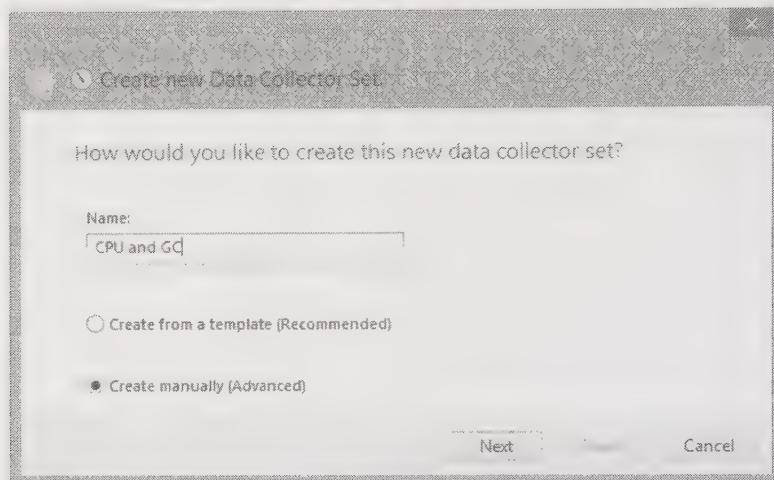


Figure 1.9. Data Collector Set configuration dialog box for setting up regular counter collections.

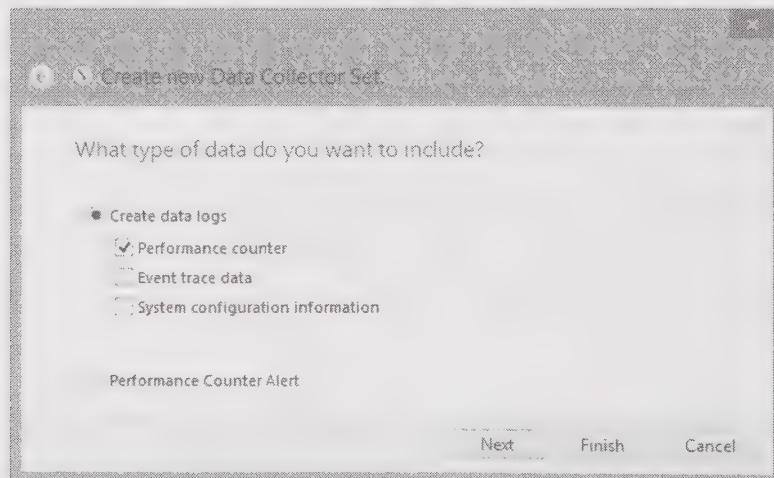


Figure 1.10. Specify the type of data you want to store.

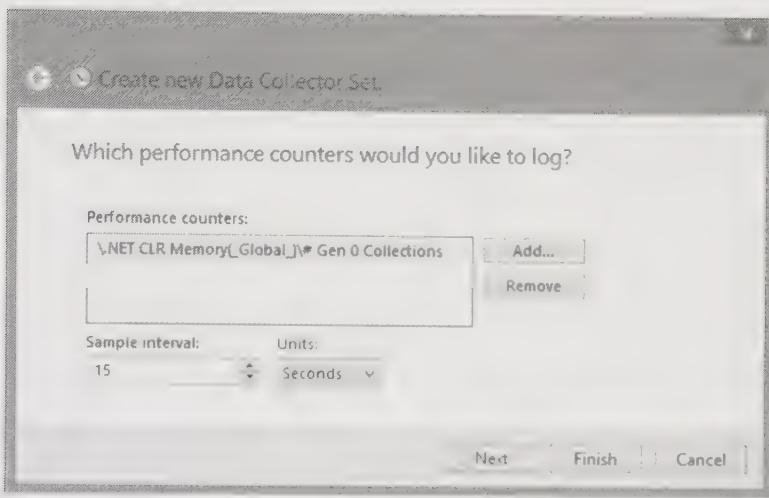


Figure 1.11. Select the counters to collect.

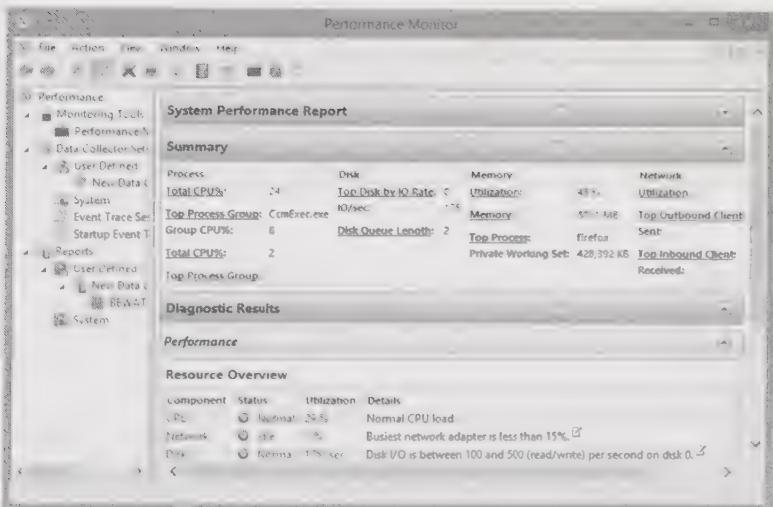


Figure 1.12. A saved report file. Use the toolbar buttons to change the view to a graph of the captured counter data.

It is likely that everything you will need to do with performance counters can be done using the functionality described here, but if you want to take programmatic control or create your own counters, see Chapter 7 for details. You should consider performance counter analysis a baseline for all performance work on your application.

ETW Events

Event Tracing for Windows (ETW) is one of the fundamental building blocks for all diagnostic logging in Windows, not just for performance. This section will give you an overview of ETW and Chapter 8 will teach you how to create and monitor your own events.

Events are produced by providers. For example, the CLR contains the Runtime provider that produces most of the events we are interested in for this book. There are providers for nearly every subsystem in Windows, such as the CPU, disk, network, firewall, memory, and many, many more. The ETW subsystem is extremely efficient and can handle the enormous volume of events generated, with minimal overhead.

Each event has some standard fields associated with it, like event level (informational, warning, error, verbose, and critical) and keywords. Each provider can define its own keywords. The CLR's Runtime provider has keywords for things like GC, JIT, Security, Interop, Contention, and more. Keywords allow you to filter the events you would like to monitor.

Each event also has a custom data structure defined by its provider that describes the state of some behavior. For example, the Runtime's GC events will mention things like the generation of the current collection, whether it was background, and so on.

What makes ETW so powerful is that, since most components in Windows produce an enormous number of events describing nearly every aspect of an application's operation, at every layer, you can do the bulk of performance analysis with ETW events only.

Many tools can process ETW events and give specialized views. In fact, starting in Windows 8, all CPU profiling is done using ETW events.

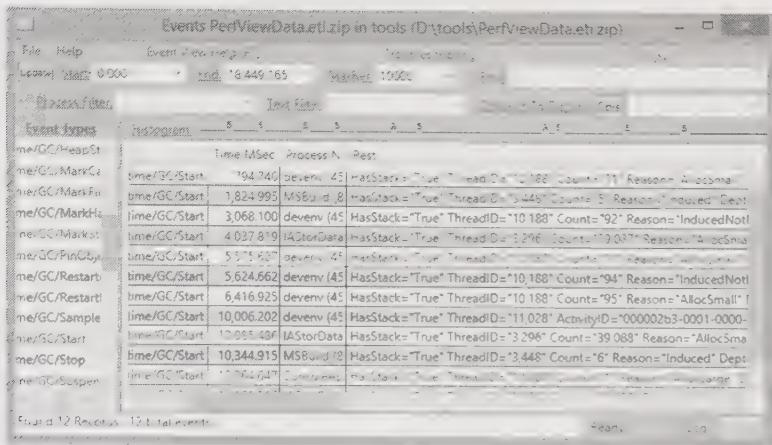


Figure 1.13. A list of all GC Start events taken in a 60-second trace. Notice various pieces of data associated with the event, such as the Reason and Depth.

To see a list of all the ETW providers registered on your system, open a command prompt and type:

```
logman query providers
```

This will produce a large amount of output similar to the following:

Provider	GUID
.NET Common Language Runtime	{E13C0D23-CCBC-4E12-931B...}
ACPI Driver Trace Provider	{DAB01D4D-2D48-477D-B1C3...}
Active Directory Domain Services: SAM	{8E598056-8993-11D2-819E...}
Active Directory: Kerberos Client	{BBA3ADD2-C229-4CDB-AE2B...}
Active Directory: NetLogon	{F33959B4-DBEC-11D2-895B...}
ADODB.1	{04C8A86F-3369-12F8-4769...}
ADOMD.1	{7EA56435-3F2F-3F63-A829...}
Application Popup	{47BFA2B7-BD54-4FAC-B70B...}
Application-Addon-Event-Provider	{A83FA99F-C356-4DED-9FD6...}
...	

You can also get details on the keywords for a specific provider:

```
D:\>logman query providers "Windows Kernel Trace"
```

Provider	GUID	
Windows Kernel Trace	{9E814AAD-3204-11D2-9A82...}	
Value	Keyword	Description
0x0000000000000001	process	Process creations/deletions
0x0000000000000002	thread	Thread creations/deletions
0x0000000000000004	img	Image load
0x0000000000000008	proccntr	Process counters
0x0000000000000010	cswitch	Context switches
0x0000000000000020	dpc	Deferred procedure calls
0x0000000000000040	isr	Interrupts
0x0000000000000080	syscall	System calls
0x0000000000000100	disk	Disk IO
0x0000000000000200	file	File details
0x0000000000000400	diskinit	Disk IO entry
0x0000000000000800	dispatcher	Dispatcher operations
0x0000000000001000	pf	Page faults
0x0000000000002000	hf	Hard page faults
0x0000000000004000	virtalloc	Virtual memory allocations
0x00000000000010000	net	Network TCP/IP
0x00000000000020000	registry	Registry details
0x000000000000100000	alpc	ALPC
0x000000000000200000	splitio	Split IO
0x00000000000800000	driver	Driver delays
0x00000000001000000	profile	Sample based profiling
0x00000000002000000	fileioccompletion	File IO completion
0x00000000004000000	fileio	File IO

Unfortunately, there is no good online resource to explain which events exist in the various providers. Some common ETW events for all Windows processes include those in the Windows Kernel Trace category:

- Memory/Hard Fault
- DiskIO/Read
- DiskIO/Write
- Process/Start
- Process/Stop
- TcpIp/Connect
- TcpIp/Disconnect
- Thread/Start
- Thread/Stop

To see other events from this provider or others, you can collect ETW events and examine them yourself.

Throughout the book, I will mention the important events you should pay attention to in an ETW trace, particularly from the CLR Runtime provider. For the complete CLR ETW documentation, you can visit <https://docs.microsoft.com/dotnet/framework/performance/etw-events-in-the-common-language-runtime>.

PerfView

Many tools can collect and analyze ETW events, but PerfView, originally written by Microsoft .NET performance architect (and writer of this book's Foreword) Vance Morrison, is one of the best for its sheer power. The previous screen shot of ETW events is from this tool.

PerfView is built upon an ETW processing engine called TraceEvent, which you can reuse yourself (See Chapter 8). But PerfView's real utility lies in its extremely powerful stack grouping and folding mechanism that lets you drill into events at multiple layers of abstraction.

While other ETW analysis tools can be useful, I often prefer PerfView for a few reasons:

1. It requires no installation so it is easy to run on any computer.
2. It is extremely configurable and easily scriptable.
3. You can pick which events to capture at a very granular level, which allows you, for example, to take hours-long traces of just a few categories of events.
4. It generally causes very little impact to the machine or processes it monitors.
5. It has unparalleled analysis capabilities through its extensive stack grouping and folding capability.
6. You can customize PerfView with extensions for your own custom analysis that take advantage of the built-in stack grouping and folding functionality.
7. It has integrated source browsing, including the source of the .NET Framework.
8. Sophisticated analysis of asynchronous calls that use the Task Parallel Library.
9. Support for IIS and ASP.NET.

Here are some common questions that I routinely answer using PerfView:

- Where is my CPU usage going?

- Who is allocating the most memory?
- What types are being allocated the most?
- What is causing my Gen 2 garbage collections?
- How long is the average Gen 0 collection?
- How much JITting is my code doing?
- Which locks are most contentious?
- What does my managed heap look like?

To collect and analyze events using PerfView follow these basic steps:

1. From the Collect menu, select the Collect menu item.
2. From the resulting dialog, specify the options you need.
 - (a) Expand Advanced Options to narrow down the type of events you want to capture.
 - (b) Check No V3.X NGEN Symbols if you are not using .NET 3.5.
 - (c) Optionally specify Max Collect Sec to automatically stop collection after the given time.
3. Click the Start Collection button.
4. If not using a Max Collect Sec value, click the Stop Collection button when done.
5. Wait for the events to be processed.
6. Select the view to use from the resulting tree.

During event collection, PerfView captures ETW events for all processes. You can filter events per-process after the collection is complete.

Collecting events is not free. Certain categories of events are more expensive to collect than others. For example, a CPU profile generates a huge number of events, so you should keep the profile time very limited (around a minute or two) or you could end up with multi-gigabyte files that you cannot analyze.

PerfView Interface and Views

Most views in PerfView are variations of a single type, so it is worth understanding how it works.

PerfView is mostly a stack aggregator and viewer. When you record ETW events, the stack for each event is recorded. PerfView analyzes these stacks and shows them to you in a grid that is common to CPU, memory allocation, lock contention, exceptions thrown, and most other types of events. The principles you learn while doing one type of investigation will apply to other types, since the stack analysis is the same.

You also need to understand the concepts of grouping and folding. Grouping turns multiple sources into a single entity. For example, there are multiple .NET Framework DLLs and which DLL a particular function is in is not usually interesting for profiling. Using grouping, you can define a grouping pattern, such as “System.*!=>LIB”, which coalesces all System.*.dll assemblies into a single group called LIB. This is one of the default grouping patterns that PerfView applies. If you wanted to, for example, collapse all method calls in the `TimeZoneInfo` class, you could have a group defined as:

```
“mscorlib.ni!System.TimeZoneInfo*->TIMEZONE”
```

This will cause TIMEZONE to appear throughout your stack in the place of any `TimeZoneInfo` methods.

Folding allows you to hide some of the irrelevant complexity of the lower layers of code by counting its cost in the nodes that call it. As a simple example, consider where memory allocations occur – always via some internal CLR method invoked by the `new` operator. What you really want to know is which types are most responsible for those allocations. Folding allows you to attribute those underlying costs to their parents, code which you can actually control. For example, in most cases you do not care about which internal operations are taking up time inside `String.Format`; you really care about what areas of your code are calling `String.Format` in the first place. PerfView can fold those operations into the caller to give you a better picture of your code’s performance.

Folding patterns can use the groups you defined for grouping. So, for example, you can just specify a folding pattern of “LIB” which will ensure that all methods in `System.*` are attributed to their caller outside of `System.*`.

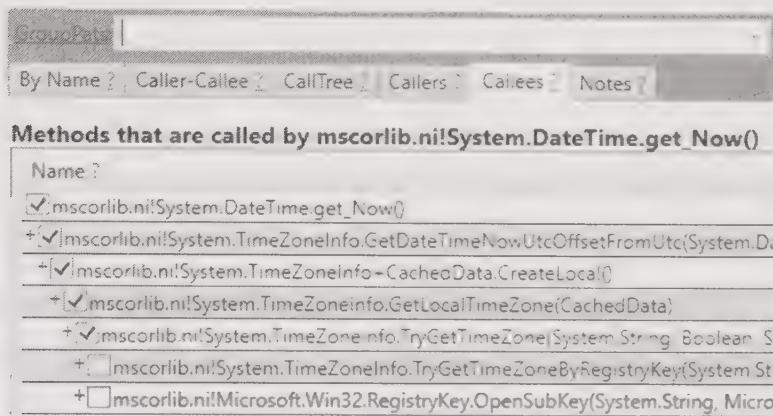


Figure 1.14. Without a grouping, there are multiple layers of calls inside the `TimeZoneInfo` class.

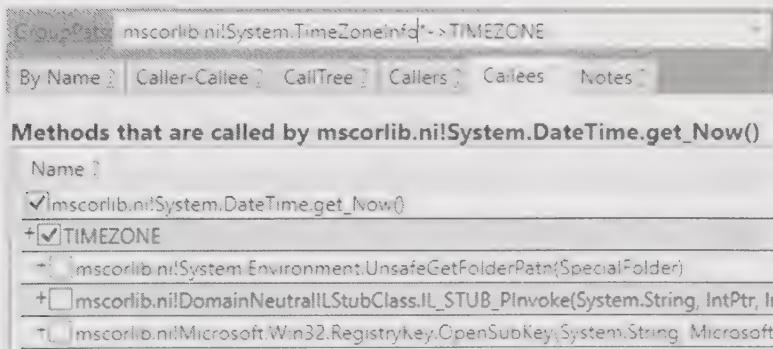


Figure 1.15. By grouping these, you can hide their details and pretend that it is all just a single frame in the stack.

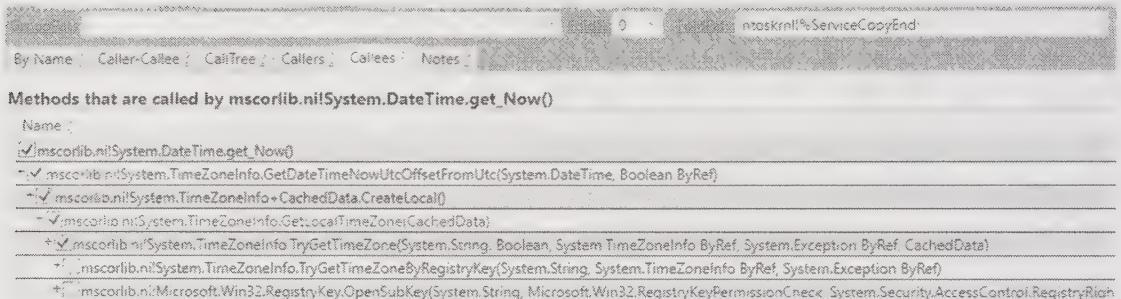


Figure 1.16. The call to `DateTime.Now` includes a deep chain of `TimeZoneInfo` method calls. Without folding, these can get a little noisy.

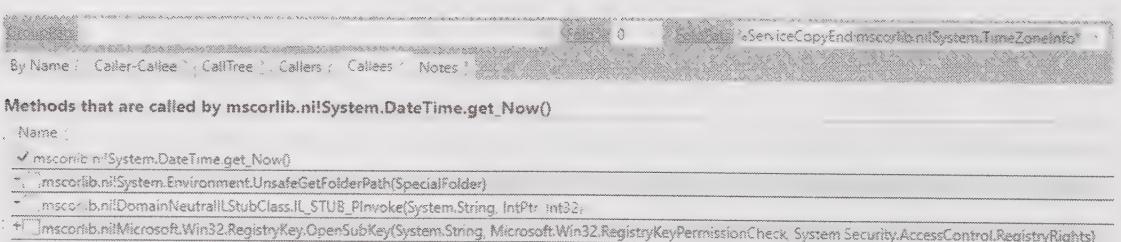


Figure 1.17. By folding the pattern “`mscorib.ni!System.TimeZoneInfo*`”, all of the cost of those methods will be counted as the cost of calling `DateTime.Now`

The user interface of the stack viewer needs some brief explanation as well.

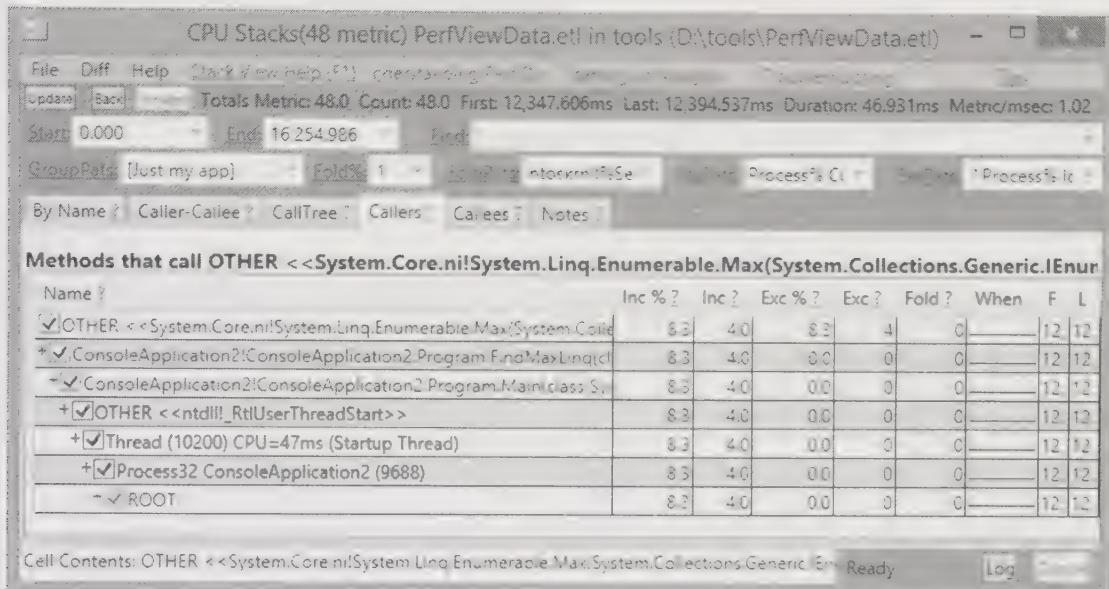


Figure 1.18. A typical stack view in PerfView. The UI contains many options for filtering, sorting, and searching.

Controls at the top allow you to organize the stack view in multiple ways. Here is a summary of their usage, but you can click on the ? in the column headers to bring up a help file that gives you more details.

- **Start:** Start time (in microseconds) which you want to examine.
- **End:** End time (in microseconds) which you want to examine.
- **Find:** Text to search for.
- **GroupPats:** A semi-colon-delimited list of grouping patterns.
- **Fold%:** Any stack that takes less than this percentage will be folded into its parent.
- **FoldPats:** A semi-colon-delimited list of folding patterns.

- **IncPats:** Stacks must have this pattern to be included in the analysis. This usually contains the process name.
- **ExcPats:** Exclude anything with this pattern from analysis. By default, this has just the Idle process.

There are a few different view tabs:

- **By Name:** Shows every node, whether type, method, or group. This is good for bottom-up analysis.
- **Caller-Callee:** Focuses on a single node, showing you callers and callees of that node.
- **CallTree:** Shows a tree of all nodes in the profile, starting at ROOT. This works well for doing top-down analysis.
- **Callers:** Shows you all callers of a particular node.
- **Callees:** Shows you all called methods of a particular node.
- **Notes:** Allows you to save notes on your investigation in the ETL files themselves.

In the grid view, there are a number of columns. Click on the column names to bring up more information. Here is a summary of the most important columns:

- **Name:** The type, method, or customized group name.
- **Exc %:** Percentage of exclusive cost. For memory traces, it is the amount of memory attributed to this type/method only. For CPU traces, it is the amount of CPU time attributed to this method.
- **Exc:** The sum of the sampled metric in just this node, excluding child nodes. For memory traces, the number of bytes attributed to this node exclusively. For CPU traces, the amount of time (in milliseconds) spent here.

- **Exc Ct:** Number of samples exclusively in this node.
- **Inc %:** Percentage of cost for this type/method and all its children. This is always at least as big as Exc %.
- **Inc:** Cost of this node, including all children. For CPU usage, this is the amount of CPU time spent in this node plus all of its children.
- **Inc Ct:** Number of samples on this node and all its children.

In the chapters that follow, I will give instructions for solving specific problems with various types of performance investigations. A complete overview of PerfView would be worth a book on its own, or at least a very detailed help file which just so happens to come with PerfView. I strongly encourage you to read this manual once you have gone through a few simple analyses.

It may seem like PerfView is mostly for analyzing memory or CPU, but do not forget that it is really just a generic stack aggregation program, and those stacks can come from any ETW event. It can analyze your sources of lock contention, disk I/O, or any arbitrary application event with the same grouping and folding power.

CLR Profiler

CLR Profiler is a possible alternative to PerfView's memory analysis capabilities if you want a graphical representation of the heap and relationships between objects. CLR Profiler can show you a wealth of detail. For example:

- Visual graph of what the program allocates and the chain of methods that led to the allocation.
- Histograms of allocated, relocated, and finalized objects by size and type.
- Histogram of objects by lifetime.
- Timeline of object allocations and garbage collections, showing the change in the heap over time.

- Graphs of objects by their virtual memory address, which can show fragmentation quite easily.

I rarely use CLR Profiler because of some of its limitations and age, but it is still occasionally useful. It has unique visualizations that no other free tool currently matches. It comes with 32-bit and 64-bit binaries as well as documentation and the source code.

The basic steps to get a trace are:

1. Pick the correct version to run: 32-bit or 64-bit, depending on your target program. You cannot profile a 32-bit program with the 64-bit profiler or vice-versa.
2. Check the Profiling active box.
3. Optionally check the Allocations and Calls boxes.
4. If necessary, go to the File | Set Parameters... menu option to set options like command line parameters, working directory, and log file directory.
5. Click the Start Application button
6. Browse to the application you want to profile and click the Open button.

This will start the application with profiling active. When you are done profiling, exit the program, or select Kill Application in CLR Profiler. This will terminate the profiled application and start processing the capture log. This processing can take quite a while, depending on the profile duration. (I have seen it take over an hour before.)

While profiling is going on, you can click the “Show Heap now” button in CLR Profiler. This will cause it to take a heap dump and open the results in a visual graph of object relationships. Profiling will continue uninterrupted, and you can take multiple heap dumps at different points.

When it is done, you will see the main results screen.

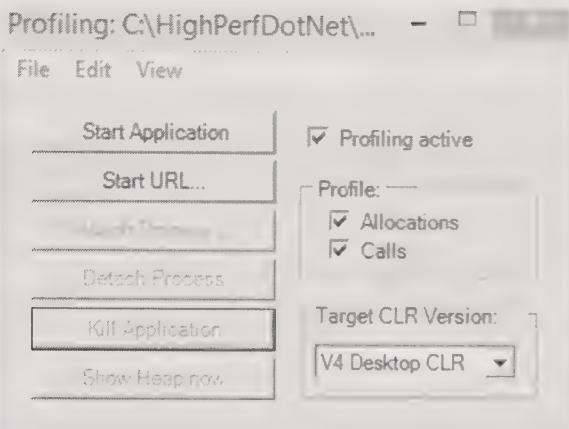


Figure 1.19. CLR Profiler's main window.

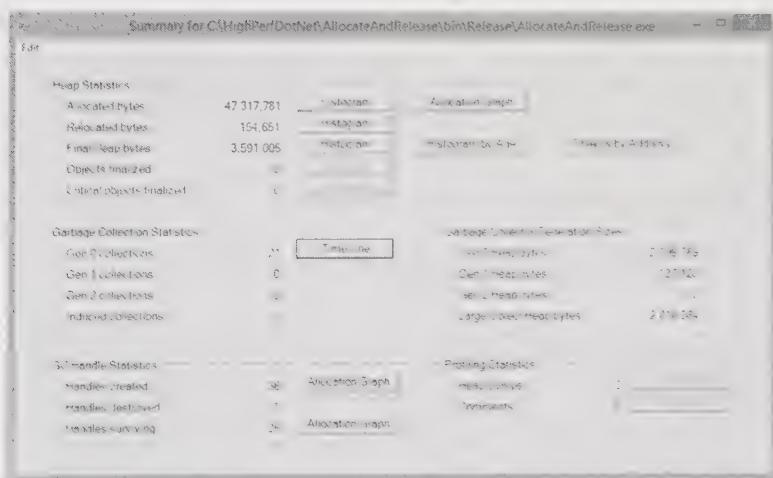


Figure 1.20. CLR Profiler's Results Summary view, showing you the data it collected during the trace.

From this screen, you can access different visualizations of heap data. Start with the Allocation Graph and the Time Line to see some of the essential capabilities. As you become comfortable analyzing managed code, the histogram views will also become an invaluable resource.

Note

While CLR Profiler is generally great, I have had a few major problems with it. First, it is a bit finicky. If you do not set it up correctly before starting to profile, it can throw exceptions or die unexpectedly. For example, I always have to check the Allocations or Calls boxes before I start profiling if I want to get any data at all. You should completely disregard the Attach to Process button, as it does not seem to work reliably. CLR Profiler does not seem to work well for truly huge applications with enormous heaps or a large number of assemblies. If you find yourself having trouble, PerfView may be a better solution because of its polish and extreme customizability through very detailed command-line parameters that allow you to control nearly all aspects of its behavior. Your mileage may vary. On the other hand, CLR Profiler comes with its own source code so you can fix it!

Windows Performance Analyzer

The Windows Assessment and Deployment Kit (Windows ADK, also part of the Windows SDK) contains a number of tools that aid in deploying operating systems and applications to machines. Inside it are a pair of tools called Windows Performance Recorder and Windows Performance Analyzer. These tools process ETW events in the same manner as PerfView. However, Windows Performance Analyzer excels in displaying hardware and operating system level information. It can display .NET events as well, but it is not as convenient as PerfView.

To capture a trace, invoke Windows Performance Recorder and start capturing.

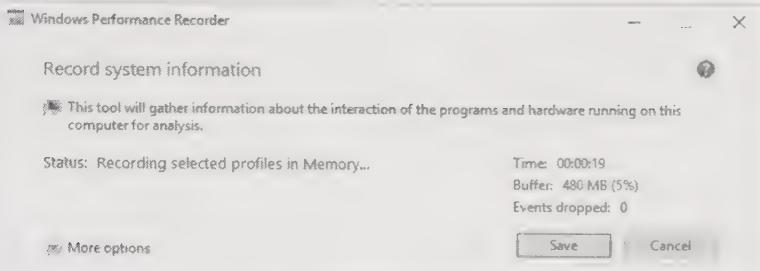


Figure 1.21. The main interface of Windows Performance Recorder. Click More Options to customize what kind of events are captured.

After you are done capturing events, click the Save button, which will bring up an interface for you to provide more details, while WPR processes the captured data in the background.

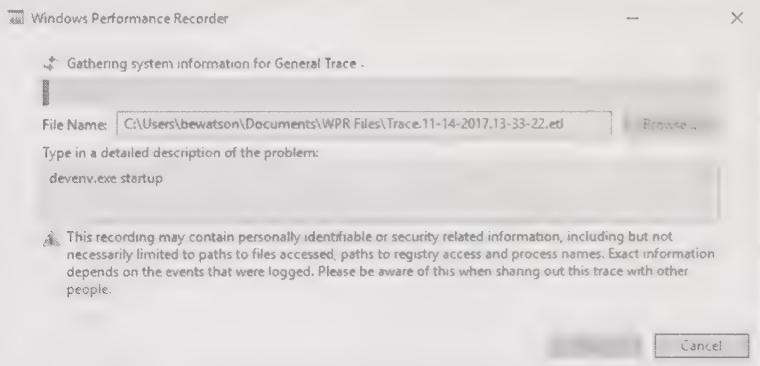


Figure 1.22. It can take a few minutes for this process to complete.

The capture data file can be opened in any tool that can analyze ETW events, but there is a convenient button to open it directly in Windows Performance Analyzer.

Windows Performance Analyzer shows you a list of resource categories along the left-hand side. Double-clicking them opens up a detailed view with a graph and table with details suitable for that resource. For example, details for memory usage will show you different categories of memory usage, such as active vs committed memory, paged pool, private pages, and more.

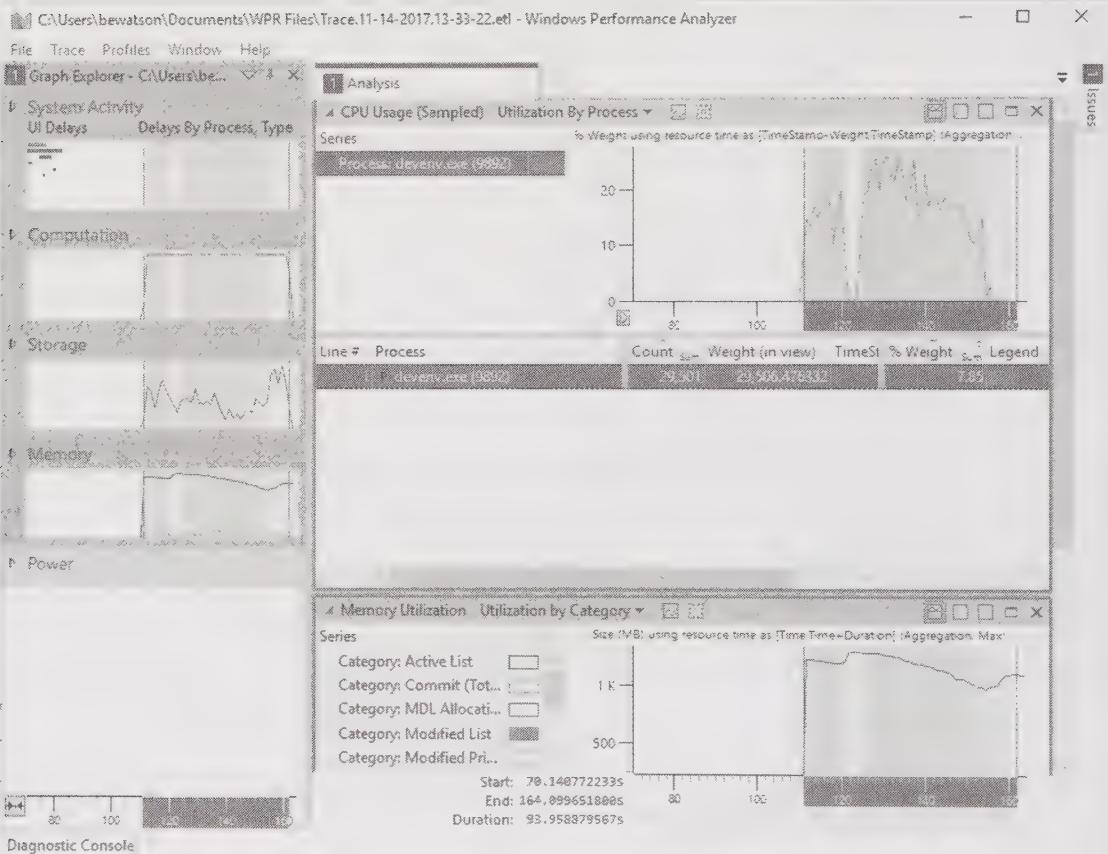


Figure 1.23. Windows Performance Analyzer's main interface, showing captured OS and hardware metrics.

Because this tool focuses more on general operating system resource usage issues, rather than .NET, I will not discuss it further in this book, but it is a useful tool to keep in mind when you are dealing with some classes of performance problems.

WinDbg

WinDbg is a general-purpose Windows Debugger distributed for free by Microsoft. If you are used to using Visual Studio as your main debugger, using this bare-bones, text-only debugger may seem daunting. Do not let it be. Once you learn a few commands, you will feel comfortable and after a while, you will rarely use Visual Studio for debugging except during active development.

WinDbg is far more powerful than Visual Studio and will let you examine your process in many ways you could not otherwise. It is also lightweight and more easily deployable to production servers or customer machines. In these situations, it is in your best interest to become familiar with WinDbg. By itself, however, WinDbg is not that interesting for managed code. To work with managed processes effectively, you will need to use .NET's SOS extensions, which ship with each version of the .NET Framework. A very handy SOS reference cheat sheet is located at <https://docs.microsoft.com/dotnet/framework/tools/sos-dll-sos-debugging-extension>. You can also use SOS.dll from Visual Studio, but this is not as straightforward, and there are other benefits to becoming familiar with WinDbg, so I will cover that scenario.

With WinDbg and SOS together, you can quickly answer questions such as these:

- How many of each object type are on the heap, and how big are they?
- How big are each of my heaps and how much of them is free space (fragmentation)?
- What objects stick around through a garbage collection?
- Which objects are pinned?

- Which threads are taking the most CPU time? Is one of them stuck in an infinite loop?

WinDbg is not usually my first tool (that is often PerfView), but it is often my second or third, allowing me to see things that other tools will not easily show. For this reason, I will use WinDbg extensively throughout this book to show you how to examine your program's operation, even when other tools do a quicker or better job. (Do not worry; I will also cover those tools.)

Do not be daunted by the text interface of WinDbg. Once you use a few commands to look into your process, you will quickly become comfortable and appreciative of the speed with which you can analyze a program. The chapters in this book will add to your knowledge little by little with specific scenarios.

To get WinDbg, you must install the Windows SDK. You can choose to install only the debuggers if you wish.

To get started with WinDbg, do a simple tutorial with a sample program. The program will be basic enough - a straightforward, easy-to-debug memory leak. You can find it in the accompanying source code in the MemoryLeak project (available at <http://www.writinghighperf.net>).

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace MemoryLeak
{
    class Program
    {
        static List<string> times = new List<string>();

        static void Main(string[] args)
        {
            Console.WriteLine("Press any key to exit");
            while (!Console.KeyAvailable)
            {
                times.Add(DateTime.Now.ToString());
                Console.Write('.');
                Thread.Sleep(10);
            }
        }
    }
}
```

```

    }
}
}
}
```

Startup this program and let it run for a few minutes.

Run WinDbg from where you installed it. It should be in the Start Menu if you installed it via the Windows SDK. Take care to run the correct version, either x86 (for 32-bit processes) or x64 (for 64-bit processes). Go to File | Attach to Process (or hit F6) to bring up the Attach to Process dialog.

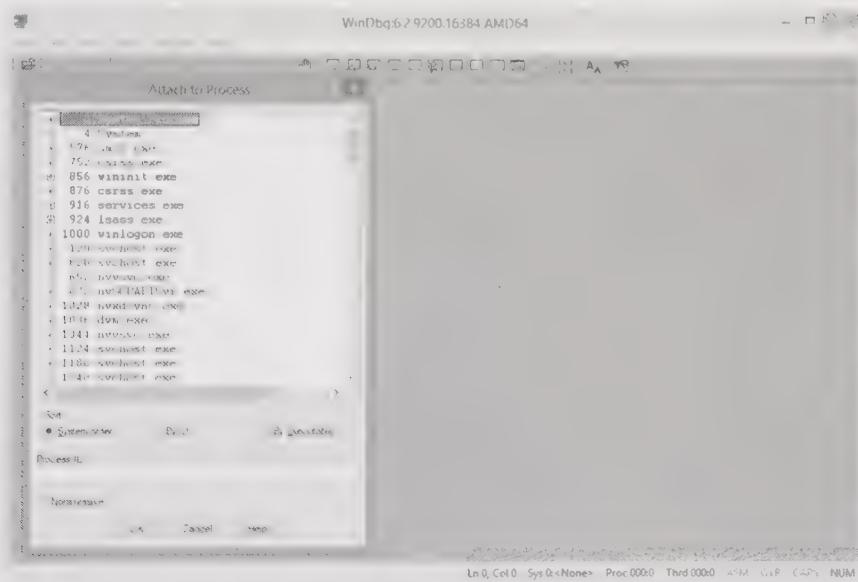


Figure 1.24. WinDbg's Attach to Process screen.

From here, find the MemoryLeak process. (It may be easier to check the By Executable sort option.) Click OK.

WinDbg will suspend the process (This is important to know if you are debugging a live production process!) and display any loaded modules. At this point, it will be waiting for your command. The first thing you usually want to do is load the CLR debugging extensions. Enter this command:

```
.loadby sos clr
```

If it succeeds, there will be no output.

If you get an error message that says “Unable to find module ‘clr’ ” it most likely means the CLR has not yet been loaded. This can happen if you launch a program from WinDbg and break into it immediately. In this case, first set a breakpoint on the CLR module load:

```
sxe ld clr  
g
```

The first command sets a breakpoint on the load of the CLR module. The g command tells the debugger to continue execution. Once you break again, the CLR module should be loaded and you can now load SOS with the `.loadby sos clr` command, as described previously.

At this point, you can do any number of things. Here are some commands to try:

```
!ProcInfo
```

This prints out some general debugging information about the process as a whole, including environment variables set:

```
Environment  
=::=:::\  
=C:=C:\WINDOWS\system32  
...many, many environment variables
```

```
Process Times  
Process Started at: 2017 Nov 7 22:5:49.44  
Kernel CPU time : 0 days 00:00:00.01  
User CPU time : 0 days 00:00:00.01
```

```
Total CPU time : 0 days 00:00:00.02
```

Process Memory

WorkingSetSize:	26572 KB	PeakWorkingSetSize:	26572 KB
VirtualSize:	717972 KB	PeakVirtualSize:	717972 KB
PagefileUsage:	566560 KB	PeakPagefileUsage:	566560 KB

44 percent of memory is in use.

Memory Availability (Numbers in MB)

	Total	Avail
Physical Memory	4095	4095
Page File	4095	4095
Virtual Memory	4095	3783

More useful commands:

g

This stands for “Go” and continues execution. You cannot enter any commands while the program is running.

<Ctrl-Break>

This pauses a running program. Do this after you Go (g) to get back control.

.dump /ma d:\memorydump.dmp

This creates a full process dump to the selected file. This will allow you to debug the process’s state later, though since it is a snapshot, of course you will not be able to debug any further execution.

```
!DumpHeap -stat
```

DumpHeap shows a summary of all managed objects on the object heap, including their size (just for this object, not any referenced objects), count, and other information. If you want to see every object on the heap of type `System.String`, type `!DumpHeap -type System.String`. You will see more about this command when investigating garbage collection.

```
~*kb
```

This is a regular WinDbg command, not from SOS. It prints the current stack for all threads in the process.

To switch the current thread to a different one, use the command:

```
~32s
```

This will change the current thread to thread 32. Note that thread numbers in WinDbg are not the same as thread IDs. WinDbg numbers all the threads in your process for easy reference, regardless of the Windows or .NET thread ID.

```
!DumpStackObjects
```

You can also use the abbreviated version: `!dso`. This dumps out the address and type of each object from all stack frames for the current thread.

Note that all commands located in the SOS debugging extension for managed code are prefixed with a `!` character.

The other thing you need to do to be effective with the debugger is set your symbol path to download the public symbols for Microsoft DLLs so you can see what is going on in the system layer. Set your `NT_SYMBOL_PATH` environment variable to this string:

```
symsrv*symsrv.dll*c:\sym*http://msdl.microsoft.com/download/symbols
```

Replace c:\sym with your preferred local symbol cache path (and make sure you create the directory). With the environment variable set, both WinDbg and Visual Studio will use this path to automatically download and cache the public symbols for system DLLs. During the initial download, symbol resolution may be quite slow, but once cached, it should speed up significantly. You can also use the .symfix command to automatically set the symbol path to the Microsoft symbol server and local cache directly:

```
.symfix c:\sym
```

If you have not used WinDbg before, do not be afraid to dive in and try it out. Once you memorize a small number of commands, you will be highly productive in no time. Deep mastery of WinDbg will come with time and experience, but it is worth the journey. You can do many types of analysis in WinDbg that are very difficult or impossible to do in other debuggers. See especially Chapter 2's section for investigating memory issues for many examples of WinDbg usage.

CLR MD

After you have used WinDbg for a while and seen the power available to you, you will likely have the thought, “I wish I could access this stuff programmatically.” Thankfully, you can! Microsoft.Diagnostics.Runtime (nicknamed “CLR MD”) is an open source library available at <https://github.com/microsoft/clrmd>. It provides access to much of the functionality in SOS.dll, in a convenient, easy-to-use API. CLR MD is designed to be a fairly low level API, allowing you to easily build on top of it to provide richer functionality. In fact, some of PerfView’s functionality is built on top of CLR MD, so if PerfView is not giving you exactly what you need, you can go under the hood, so to speak, to this library, and build what you need.

In this section, I’ll provide an overview of the tool and how to use it, but specific solutions to problems will be found in the relevant sections throughout the book.

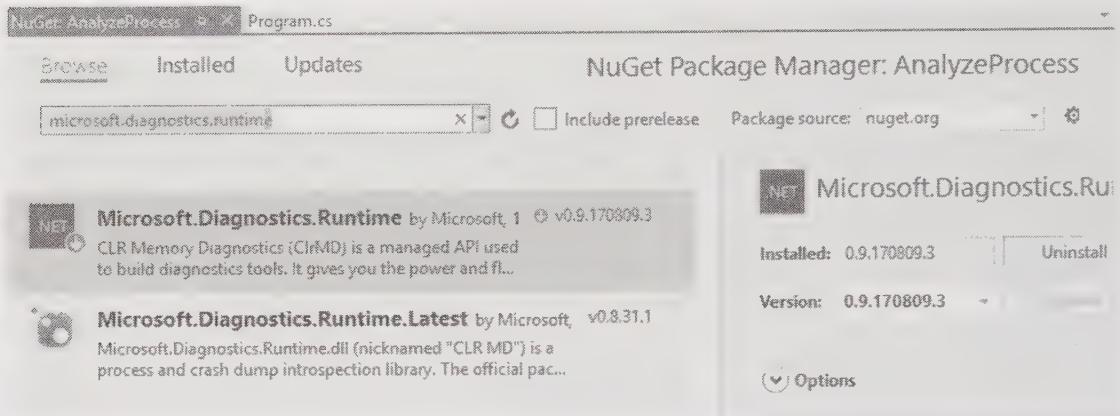


Figure 1.25. The easiest way to obtain the library is via a NuGet package from <http://NuGet.org>. You can search for either “Microsoft.Diagnostics.Runtime” or “CLR MD”.

Note

The library is very much in active development, and you will see differences between the documentation and what is currently implemented. The API may also change further.

You can use this library to both attach to live processes (as a debugger), or open heap dump files on disk. I'll show examples of both.

To attach to a live process, you just need to supply a process ID. In this example, I'm explicitly starting a new process for convenience. Most examples of CLR MD in this book will come from the AnalyzeProcess sample code project accompanying this book.

```
static void Main(string[] args)
{
    // Let's create our own process to test with
    var startInfo = new ProcessStartInfo(TargetProcessName);
    startInfo.CreateNoWindow = true;
    startInfo.WindowStyle = ProcessWindowState.Hidden;
```

```
var targetProcess = Process.Start(startInfo);
Thread.Sleep(1000);
using (DataTarget target = DataTarget.AttachToProcess(
    targetProcess.Id,
    10000, // timeout
    AttachFlag.Invasive))
{
    PrintDumpInfo(target);

    var clr = target.ClrVersions[0].CreateRuntime();
}
}

private static void PrintDumpInfo(DataTarget target)
{
    PrintHeader("Target Info");

    Console.WriteLine($"Architecture: {target.Architecture}");
    Console.WriteLine($"Pointer Size: {target.PointerSize}");
    Console.WriteLine("CLR Versions:");
    foreach(var clr in target.ClrVersions)
    {
        Console.WriteLine($"{clr.Version}");
    }
}
```

This program will print out the following information:

```
Target Info
=====
Architecture: X86
Pointer Size: 4
CLR Versions:
    v4.7.2115.00
```

The `clr` object obtained after calling `PrintDumpInfo` is the main interface to most of the interesting commands. Using it, you can, for example, iterate over every object in the heap:

```
var heap = clr.Heap;
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
    Console.WriteLine(
        $"0x{obj.Address:x} - {obj.Type.Name}" +
        $" - Generation: {generation}");
}
```

Which produces output similar to:

```
0x30ec8ac - System.Byte[] - Generation: 0
0x30ecca0 - LargeMemoryUsage.B - Generation: 1
```

In addition to the heap, you can examine code:

```
foreach(var module in clr.Modules)
{
    foreach (var type in module.EnumerateTypes())
    {
        foreach(var method in type.Methods)
        {
            Console.WriteLine(method.Name);
        }
    }
}
```

This produces output like this:

```
Main
GetNewObject
.cctor
ToString
ToString
Equals
```

You can also open crash dumps. This is slightly more complicated because you must also obtain the mscordacwks.dll file that matches the CLR version(s) present in the dump. When attaching to a live process, this is trivial because it is guaranteed to be present on the machine. With a dump from a different machine, and potentially a different version of the CLR altogether, you must obtain it from that machine or download it from the Microsoft symbol server. This code shows you how to accomplish this:

```
{  
    ...  
  
    string dacFile =  
        GetDacFile(  
            dataTarget.ClrVersions[0],  
            dataTarget);  
    var clr = dataTarget.ClrVersions[0].CreateRuntime(dacFile);  
    ...  
}  
  
private static string GetDacFile(ClrInfo clrInfo,  
                                DataTarget target)  
{  
    string location = clrInfo.LocalMatchingDac;  
    if (string.IsNullOrEmpty(location) || !File.Exists(location))  
    {  
        // try to download from symbol server  
        ModuleInfo dacInfo = clrInfo.DacInfo;  
        try  
        {  
            location = target.SymbolLocator.FindBinary(dacInfo);  
        }  
        catch (WebException)  
        {  
            return null;  
        }  
    }  
    return location;  
}
```

This method is equivalent to calling `CreateRuntime` with no arguments, but it is useful to know how to do this yourself in case you have custom needs.

You will see more examples of its power in later chapters, but a summary of some of the things it can tell you:

- Enumerate all objects in the heap, and give information such as generation, whether they are pinned, etc.
- Provide tools to find objects' roots and sizes
- Enumerate memory segments
- Iterate all methods in process
- Calculate IL and native code sizes

Note

I have seen a couple of issues when using this library to examine the code in a truly huge DLL. The APIs in Microsoft.Diagnostics.Runtime rely on internal .NET APIs that may not have the most efficient implementation. In one case, I was using a dump file to calculate how much JITting had happened in a 500 MB DLL with 80,000 types, and hundreds of thousands of methods. I hit Ctrl-Break after about 36 hours. That is the only DLL I've had issues with.

IL Analyzers

There are many free and paid products out there that can take a compiled assembly and decompile it into IL, C#, VB.NET, or any other .NET language. Some of the most popular include Reflector, ILSpy, and dotPeek, but there are others.

These tools are valuable for showing you the inner details of other people's code, something critical for good performance analysis. I use them most often to examine the .NET Framework itself when I want to see the potential performance implications of various APIs.

Converting your own code to readable IL is also valuable because it can show you many operations, such as boxing, that are not visible in the higher-level languages.

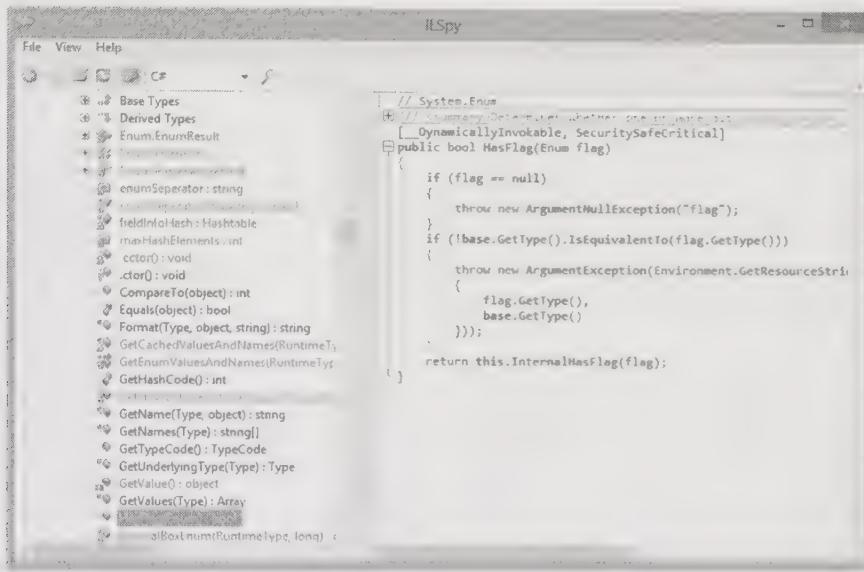


Figure 1.26. ILSpy with a decompilation of `Enum.HasFlag` in C#. Decompilers are a powerful tool for learning how 3rd-party code works and performs.

Chapter 6 discusses the .NET Framework code and encourages you to train a critical eye on every API you use. Tools like ILSpy, dotPeek, and Reflector are vital for that purpose and you will use them frequently as you become more familiar with existing code. You will often be surprised at how much work goes into seemingly simple methods. Analyzing the assemblies of other developers and companies can teach you much about good (or bad) organization, design, and coding practices.

Some other things these tools can show you:

- Assembly references
- Assembly metadata such as target framework, processor architecture
- IL code (we will make extensive use of this feature in this book)

- Size of code

Most tools also have search capability to allow you to find types, methods, fields, or code statements.

MeasureIt

MeasureIt is a handy micro-benchmark tool by Vance Morrison (the same author of PerfView). It shows the relative costs of various .NET APIs in many categories including method calls, arrays, delegates, iteration, reflection P/Invoke, and many more. It compares all the costs to calling an empty static function as a benchmark.

MeasureIt is primarily useful to show you how design choices will affect performance at an API level. For example, in the locks category, it shows you that using `ReaderWriteLock` is about four times slower than just using a regular `lock` statement.

It is easy to add your own benchmarks to MeasureIt's code. It ships with its own code packed inside itself – just run `MeasureIt /edit` to extract it. Studying this code will give you a good idea of how to write accurate benchmarks. There is a lengthy explanation in the code comments about how to do high-quality analysis, which you should pay special attention to, especially if you want to do some simple benchmarking yourself.

For example, it prevents the compiler from inlining function calls:

```
[MethodImpl(MethodImplOptions.NoInlining)]
public void AnyEmptyFunction()
{}
```

There are other tricks it uses such as working around processor caches and doing enough iterations to produce statistically significant results.

MeasureIt is handy because it has a number of built-in measurements of the CLR itself, which can give you a good idea of what the basics cost. If you are interested in benchmarking your own code, then read on to the next section.

BenchmarkDotNet

The standard in .NET benchmarking is probably the open-source project BenchmarkDotNet. This library handles many of the usual concerns about micro-benchmarking and does much more by:

- Making it easy to pick code for benchmarking with attributes.
- Generating isolated projects for each method under test.
- Automatically calculating iteration count for the required precision.
- Warming up code.
- Performing statistical analysis.
- Comparing performance across a number of different code environments, such as x86, x64, different JIT versions, GC configuration, and more.
- Analyzing CPU, garbage collection, memory allocations, JIT, and various hardware counters.

Getting started is very easy. Here is a simple example, comparing the performance of `foreach` loops on an array versus `IEnumerable`. With simple attribute decoration, you can let the library do almost all the work for you.

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System.Collections.Generic;

namespace BenchmarkTest
{
    public class LoopBenchmarks
    {
        static int[] arr = new int[100];

        public LoopBenchmarks()
        {
            for (int i = 0; i < arr.Length; i++)

```

```
        {
            arr[i] = i;
        }
    }

[Benchmark]
public int ForEachOnArray()
{
    int sum = 0;
    foreach (int val in arr)
    {
        sum += val;
    }
    return sum;
}

[Benchmark]
public int ForEachOnIEnumerable()
{
    int sum = 0;
    IEnumerable<int> arrEnum = arr;
    foreach (int val in arrEnum)
    {
        sum += val;
    }
    return sum;
}

class Program
{
    static void Main(string[] args)
    {
        var summary = BenchmarkRunner.Run<LoopBenchmarks>();
    }
}
```

You can run this yourself with the BenchmarkTest sample code.

The output ends with this:

```
Total time: 00:00:43 (43.64 sec)
```

```
// * Summary *
```

BenchmarkDotNet=v0.10.9, OS=Windows 10 Redstone 2 (10.0.15063)
 Processor=Intel Core i7-3930K CPU 3.20GHz (Ivy Bridge),
 ProcessorCount=12
 Frequency=14318180 Hz, Resolution=69.8413 ns, Timer=HPET
 [Host] : .NET Framework 4.7 (CLR 4.0.30319.42000),
 32bit LegacyJIT-v4.7.2102.0
 DefaultJob : .NET Framework 4.7 (CLR 4.0.30319.42000),
 32bit LegacyJIT-v4.7.2102.0

Method	Mean	Error	StdDev
ForEachOnArray	53.32 ns	0.2083 ns	0.1846 ns
ForEachOnIEnumerable	561.69 ns	7.2943 ns	6.8231 ns

```
// * Hints *
```

Outliers

ForEachTest.ForEachOnArray: Default -> 1 outlier was removed

```
// * Legends *
```

Mean : Arithmetic mean of all measurements

Error : Half of 99.9% confidence interval

StdDev : Standard deviation of all measurements

1 ns : 1 Nanosecond (0.000000001 sec)

```
// ***** BenchmarkRunner: End *****
```

```
// * Artifacts cleanup *
```

Notice that even for such simple code, it took a full 43 seconds to execute the benchmarks.

You can of course customize how these benchmarks work with additional configuration.

To read more, visit <http://benchmarkdotnet.org>. Add it to your project directly from Visual Studio by installing the BenchmarkDotNet NuGet package.

Code Instrumentation

The old standby of brute-force debugging via console output is still a valid scenario and should not be ignored. Rather than console output, however, I encourage you to use ETW events instead, as detailed in Chapter 8.

Performing accurate code timing is also a useful feature at times. Never use `DateTime.Now` for tracking performance data. It is just too slow for this purpose. Instead, use the `System.Diagnostics.Stopwatch` class to track the time span of small or large events in your program with extreme accuracy, precision, and low overhead.

```
var stopwatch = Stopwatch.StartNew();
...do work...
stopwatch.Stop();
TimeSpan elapsed = stopwatch.Elapsed;
long elapsedTicks = stopwatch.ElapsedTicks;
```

See Chapter 6 for more information about using times and timing in .NET.

If you want to ensure that your own benchmarks are accurate and reproducible, study the source code and documentation to MeasureIt, which highlights the best practices on this topic. It is often harder than you would expect and performing benchmarks incorrectly can be worse than doing no benchmarks at all because it will cause you to waste time on the wrong thing. It would be better to use a 3rd-party library like BenchmarkDotNet.

SysInternals Utilities

No developer, system administrator, or even hobbyist should be without this great set of tools. Originally developed by Mark Russinovich and Bryce Cogswell and now owned by Microsoft, these are tools for computer management, process inspection, network analysis, and a lot more. Here are some of my favorites:

- **ClockRes**: Shows the resolution of the system's clock (which is also the maximum timer resolution).
- **CoreInfo**: Relates logical processors to physical processors, sockets, caches, and more.
- **Diskmon**: Monitors all disk activity.
- **DiskView**: Sector-by-sector utility for hard disks.
- **Handle**: Shows which files are opened by which processes.
- **ListDLLs**: Lists loaded DLLs.
- **NTFSInfo**: Get detailed version about NTFS volumes.
- **PsInfo**: Displays OS, disk, user, and software information about a system.
- **ProcDump**: A highly configurable process dump creator.
- **Process Explorer**: A much better Task Manager, with a wealth of detail about every process.
- **Process Monitor**: Monitor file, registry, and process activity in real-time.
- **RAMMap**: Analyze the physical memory usage of the entire system.
- **SDelete**: A secure file delete utility.
- **Strings**: Searches binaries for strings.
- **VMMMap**: Analyze a process's address space.

There are dozens more. You can download this suite of utilities (individually or as a whole) from <https://docs.microsoft.com/sysinternals/>.

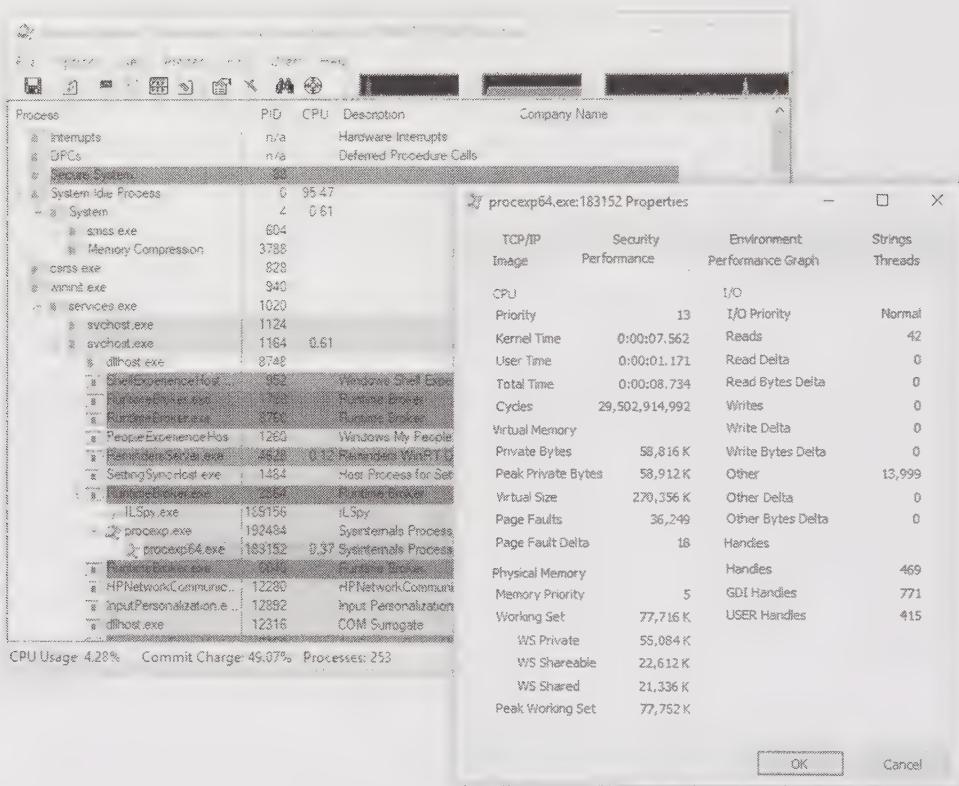


Figure 1.27. Process Explorer is a highly advanced version of Task Manager that gives you an extreme amount of detail about each process, as well as the relationships among processes.

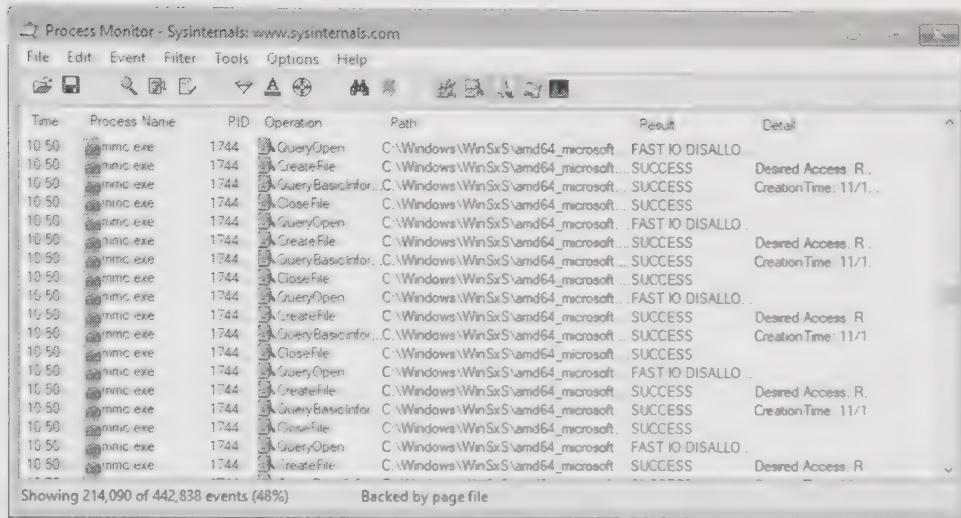


Figure 1.28. Process Monitor shows a live trace of file, registry, process, thread, and network events for the whole system. It can be useful, for example, to find out whether a process is reading from a specific file and when.

Database

The final performance tool is a rather generic one: a simple database—something to track your performance over time. The metrics you track are whatever is relevant to your project, and the format does not have to be a full-blown SQL Server relational database (though there are certainly advantages to such a system). It can be a collection of reports stored over time in an easily readable format, or just CSV files with labels and values. The point is that you should record it, store it, and build the ability to report from it.

When someone asks you if your application is performing better, which is the better answer?

Yes.

Or:

In the last 6 months, we have reduced CPU usage by 50%, memory consumption by 25%, and request latency by 15%. Our GC rate is down to one in every 10 seconds (it used to be every second!), and our startup time is now dominated entirely by configuration loading (35 seconds).

As mentioned earlier, bragging about performance gains is so much better with solid data to back it up!

Other Tools

You can find many other tools. There are plenty of static code analyzers, ETW event collectors and analyzers, assembly decompilers, performance profilers, and much more.

You can consider the list presented in this chapter as a starting point, but understand that you can do significant work with just these tools. Sometimes an intelligent visualization of a performance problem can help, but you will not always need it.

You will also discover that as you become more familiar with technologies like Performance Counters or ETW events, it is easy to write your own tools to do custom reporting or intelligent analysis. Many of the tools discussed in this book are automatable to some degree.

Measurement Overhead

No matter what you do, there is going to be some overhead from measuring your performance. CPU profiling slows your program down somewhat, performance counters will require memory and/or disk space. ETW events, as fast as they are, are not free.

You will have to monitor and optimize this overhead in your code just like all other aspects of your program. Then decide whether the cost of measurement in some scenarios is worth the performance hit you will pay.

If you cannot afford to measure all the time, then you will have to settle for some kind of profiling. As long as it is often enough to catch issues, then it is likely fine. However, do not underestimate the people cost of manual performance measurement – often, this can add up to a much higher cost than building a system that can automatically perform measurement for you.

You could also have “special builds” of your software, but this can be a little dangerous. You do not want these special builds to morph into something that is unrepresentative of the actual product.

As with many things in software, there is a balance you will have to find between having all the data you want and having optimal performance.

Summary

The most important rule of performance is ***Measure, Measure, Measure!***

Know what metrics are important for your application. Develop precise, quantifiable goals for each metric. Average values are good, but pay attention to percentiles as well, especially for high-availability services. Ensure that you include good performance goals in the design up front and understand the performance implications of your architecture. Optimize the parts of your program that have the biggest impact first. Focus on macro-optimizations at the algorithmic or systemic level before moving on to micro-optimizations. When you are unsure about the performance of an algorithm, utilize benchmarking frameworks to test them.

Have a good foundation of performance counters and ETW events for your program. For analysis and debugging, use the right tools for the job. Learn how to use the most powerful tools like WinDbg and PerfView to solve problems quickly.

Chapter 2

Memory Management

Garbage collection, and memory management in general, will be the first and last things you work on. It is the apparent source of the most obvious performance problems, those that are quickest to fix, and will be something that you need to constantly monitor to keep in check. I say “apparent source” because as we will see, many problems are actually due to an incorrect understanding of the garbage collector’s behavior and expectations. You need to think of memory performance just as much as CPU performance. This is also true for unmanaged code performance, but in .NET it is a little more prominent, as well as easier to deal with. It is so fundamental to smooth .NET operation, that the most significant chunk of this book’s content is dedicated to just this topic.

Many people get very nervous when they think of the overhead garbage collection can cause. Once you understand it, though, it becomes straightforward to optimize your program for its operation. In the Introduction, you saw that the garbage collector can actually give you better overall heap performance in many cases because it deals with allocation and fragmentation better. In many ways, .NET’s memory management strategy, including the garbage collector, can actually be a benefit to your application, not a drawback.

I am covering garbage collection at the beginning of the book because so many of the concepts that come later will relate back to this chapter. Understanding the effect your program has on the garbage collector is so fundamental to achieving good performance, that it affects nearly everything else.

Memory Allocation

There are significant differences between how typical native heaps work and how the CLR's garbage collected heaps work. The native heap in Windows maintains free lists to know where to put new allocations. Many long-running native code applications struggle with fragmentation. Time spent in memory allocation gradually increases as the allocator spends more and more time traversing the free lists looking for an open spot. Memory use continues to grow and, inevitably, the process will need to be restarted to begin the cycle anew. Some native programs deal with this by replacing the default implementation of `malloc` with custom allocation schemes that work hard to reduce this fragmentation. Windows also provides low-fragmentation heaps, which the CLR uses internally.

In .NET, memory allocation is trivial because it usually happens at the end of a memory segment and is not much more than a few instructions, such as additions, decrements, and a comparison in the normal case. In these simple cases, there are no free lists to traverse and little possibility of fragmentation. GC heaps can actually be more efficient because objects allocated together in time tend to be near one another on the heap, improving locality.

In the default allocation path, a small code stub will check the desired object's size against the space remaining in a small allocation buffer. As long as the allocation fits, it is extremely fast and has no contention. Once the allocation buffer is exhausted, the GC allocator will take over and find a spot for the object (this may involve the use of free lists). Then a new allocation buffer will be reserved for future allocation requests.

The assembly code for this process is only a handful of instructions and useful to examine.

The C# to demonstrate this is just a simple allocation:

```
class MyObject {
    int x;
    int y;
    int z;
}
```

```
static void Main(string[] args)
{
    var x = new MyObject();
}
```

Here is the breakdown of the calling code for the allocation:

```
; Copy method table pointer for the class into
; ecx as argument to new()
; You can use !dumpmt to examine this value.
mov    ecx,3F3838h

; Call new
call   003e2100

; Copy return value (address of object) into a register
mov    edi,eax
```

Here is the actual allocation:

```
; NOTE: Most code addresses removed for formatting reasons.
;

; Set eax to value 0x14, the size of the object to
; allocate, which comes from the method table
mov    eax,dword ptr [ecx+4] ds:002b:003f383c=00000014

; Put allocation buffer information into edx
mov    edx,dword ptr fs:[0E30h]

; edx+40 contains the address of the next available byte
; for allocation. Add that value to the desired size.
add    eax,dword ptr [edx+40h]

; Compare the intended allocation against the
; end of the allocation buffer.
cmp    eax,dword ptr [edx+44h]

; If we spill over the allocation buffer,
; jump to the slow path
ja     003e211b
```

```
; update the pointer to the next free
; byte (0x14 bytes past old value)
mov    dword ptr [edx+40h],eax

; Subtract the object size from the pointer to
; get to the start of the new obj
sub    eax,dword ptr [ecx+4]

; Put the method table pointer into the
; first 4 bytes of the object.
; eax now points to new object
mov    dword ptr [eax],ecx

; Return to caller
ret

; Slow Path - call into CLR method
003e211b  jmp clr!JIT_New (71763534)
```

In summary, this involves one direct method call and only nine instructions in the helper stub. That is hard to beat.

If you are using some configuration options such as server GC, then there is not even contention for the fast or the slow allocation path because there is a heap for every processor. .NET trades the simplicity in the allocation path for more complexity during de-allocation, but you do not have to deal with this complexity directly. You just need to learn how to optimize for it, which is what you will learn how to do in this chapter.

There are some ways to force the allocator to go down the slow path, however. If the allocation buffer is not large enough or the end of the segment has been reached, then the slow path will be called. In addition, if the object being allocated implements a finalizer, then the garbage collector needs to do more book-keeping to track the object's lifetime, thus it will call the slow path as well.

Garbage Collection Operation

The details of how the garbage collector makes decisions are continually being refined, especially as .NET becomes more prevalent in high-performance systems. The following explanation may contain details that will change in upcoming .NET versions, but the overall picture is unlikely to change drastically in the near future.

In a managed process, there are two types of heaps: unmanaged and managed. Unmanaged heaps are allocated with the `VirtualAlloc` Windows API and used by the operating system and CLR for unmanaged memory such as that for the Windows API, OS data structures, and even much of the CLR. The CLR allocates all managed .NET objects on the managed heap, also called the GC heap, because the objects on it are subject to garbage collection.

The managed heap is further divided into two types of heaps: the small object heap and the large object heap (LOH). Each one is assigned its own segments, which are blocks of memory belonging to that heap. Both the small object heap and the large object heap can have multiple segments assigned to them. The size of each segment can vary depending on your configuration and hardware platform.

Configuration	32-bit Segment Size	64-bit Segment Size
Workstation GC	16 MB	256 MB
Server GC	64 MB	4 GB
Server GC with > 4 logical processors	32 MB	2 GB
Server GC with > 8 logical processors	16 MB	1 GB

The small object heap segments are further divided into generations. There are three generations, referenced casually as gen 0, gen 1, and gen 2. Gen 0 and gen 1 are always in the same segment, but gen 2 can span multiple segments, as can the large object heap. The segment that contains gen 0 and gen 1 is called the ephemeral segment.

To start with, the small object heap is made up of one segment and the large object heap is another segment. Gen 2 and gen 1 start off at only a few bytes in size because they are empty so far.

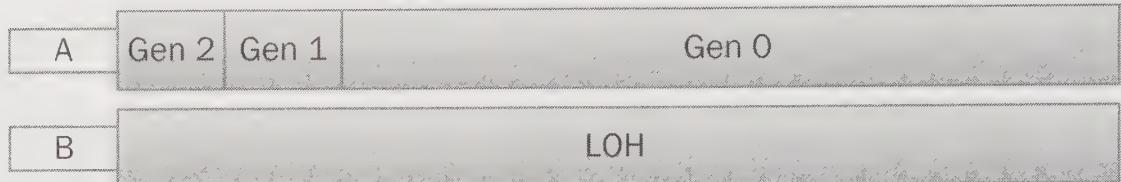


Figure 2.1. Initial heap layout.

Objects allocated on the small object heap pass through a lifetime process that needs some explanation. The CLR allocates all objects that are less than 85,000 bytes in size on the small object heap. They are always allocated in gen 0, usually at the end of the current used space. This is why allocations in .NET are extremely fast, as seen at the beginning of this chapter. If the fast allocation path fails, then the objects may be placed anywhere they can fit inside gen 0's boundaries. If it will not fit in an existing spot, then the allocator will expand the current boundaries of gen 0 to accommodate the new object. This expansion occurs at the end of the used space towards the end of the segment. If this pushes past the end of the segment, it may trigger a garbage collection. The existing gen 1 space is untouched.

For small objects (less than 85,000 bytes), objects always begin their life in gen 0. As long as they are still alive, the GC will promote them to subsequent generations each time a collection happens. Garbage collections of gen 0 and gen 1 are sometimes called ephemeral collections.

When a garbage collection occurs, a compaction may occur, in which case the GC physically moves the objects to a new location to free space in the segment. If no compaction occurs, the boundaries are merely redrawn.

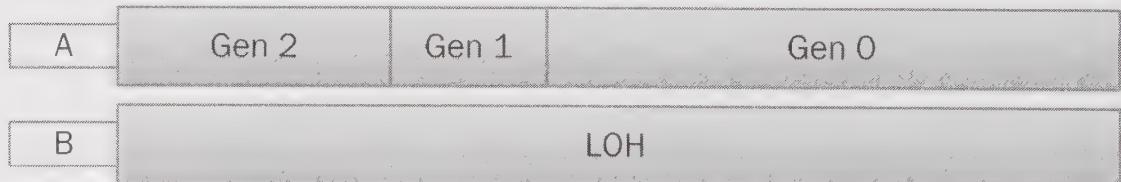


Figure 2.2. Heap layout after garbage collection.

The individual objects have not moved, but the boundary lines have.

Compaction may occur in the collection of any generation and this is a relatively expensive process because the GC must fix up all of the references to those objects so they point to the new location, which may require pausing all managed threads. Because of this expense, the garbage collector will only do compaction when it is productive to do so, based on some internal metrics.

Once an object reaches gen 2, it remains there for the remainder of its lifetime. This does not mean that gen 2 grows forever – if the objects in gen 2 finally die off and an entire segment has no live objects, then the garbage collector can return the segment to the operating system or it can just hold on to it for future use. Process working set memory is not guaranteed to drop during a collection.

So what does alive mean? If the GC can reach the object via any of the known GC roots, following the graph of object references, then it is alive. A root can be the static variables in your program, the threads which have the stacks from all running methods (thus references to local variables), strong GC handles (such as pinned handles), and the finalizer queue. Note that you may have objects that no longer have roots to them, but if the objects are in gen 2, then a gen 0 collection will not clean them up. They will have to wait for a full collection.

If gen 0 ever starts to fill up a segment and a collection cannot compact it enough, then the GC will allocate a new segment. The new segment will house a new gen 1 and gen 0 while the old segment is converted to gen 2. Everything from the old generation 0 becomes part of the new generation 1 and the old generation 1 is likewise promoted to generation 2 (which conveniently does not have to be copied).

If gen 2 continues to grow, then it can span multiple segments. The LOH can also span multiple segments. Regardless of how many segments there are, generations 0 and 1 will always exist in the same segment. This knowledge of segments will come in handy later when we are trying to figure out which objects live where on the heap.

The large object heap obeys different rules. Any object that is at least 85,000 bytes in size is allocated on the LOH automatically and does not pass through the generational model – put another way, it is allocated directly in gen 2. The only types of objects that normally exceed this size are arrays and strings. For

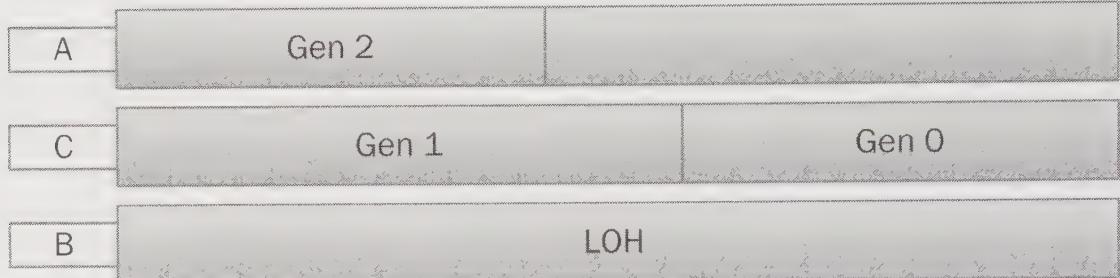


Figure 2.3. Heap layout after more allocations and collections cause new segments to be allocated.

performance reasons, the LOH is not automatically compacted during collection and is thus easily susceptible to fragmentation. However, starting in .NET 4.5.1, you can compact it on-demand. Like gen 2, if memory in the LOH is no longer needed, then it can be reclaimed for other portions of the heap, but we will see later that ideally you do not want memory on the large object heap to be garbage collected at all.

In the LOH, the garbage collector always uses a free list to determine where to best place allocated objects. We will explore some techniques in this chapter to reduce fragmentation on this heap.

Note

If you go poking around at the objects in the LOH in a debugger, you will notice that not only can the entire heap be smaller than 85,000 bytes in size, but that it can also have objects that are smaller than that size allocated on that heap. These objects are usually allocated by the CLR and you can ignore them.

A garbage collection runs for a specific generation and all generations below it. If it collects gen 1, it will also collect gen 0. If it collects gen 2, then all generations are collected, and the large object heap is collected as well. When a gen 0 or gen 1 collection occurs, the program is paused for the duration of the collection. For a gen 2 collection, portions of the collection can occur on a background thread, depending on the configuration options.

There are four phases to a garbage collection:

1. **Suspension:** All managed threads in the application are forced to pause before a collection can occur. It is worth noting that suspension can only occur at certain safe points in code, like at a `ret` instruction. Native threads are not suspended and will keep running unless they transition into managed code, at which point they too will be suspended. If you have a lot of threads, a significant portion of garbage collection time can be spent just suspending threads.
2. **Mark:** Starting from each root, the garbage collector follows every object reference and marks those objects as seen. Roots include thread stacks, pinned GC handles, and static objects.
3. **Compact:** Reduce memory fragmentation by relocating objects to be next to each other and update all references to point to the new locations. This happens on the small object heap when needed and there is no way to control it. On the large object heap, compaction does not happen automatically at all, but you can instruct the garbage collector to compact it on-demand.
4. **Resume:** The managed threads are allowed to resume.

The mark phase does not actually need to touch every object on the heap; it will only go through the target portion of the heap. For example, a gen 0 collection considers objects only from gen 0, a gen 1 collection will mark objects in both gen 0 and gen 1, and a gen 2, or full, collection, will need to traverse every live object in the heap, making it potentially very expensive.

An additional wrinkle here is that an object in a higher generation may be a root for an object in a lower generation. To track objects across all generations, the GC uses a card table that summarizes the heap with an array of bits that each represent a heap range. The bit is set “dirty” on a memory write in the corresponding range. When a collection happens, the GC will also consider any objects located in a dirty region as roots. This enables the GC to traverse only a subset of objects in the higher generation and it is not as expensive as a full collection for that generation.

There are a couple of important consequences to the behavior described above.

First, the time it takes to do a garbage collection is almost entirely dependent on the number of live objects in the collected generation, not the number of objects you allocated. This means that if you allocate a tree of a million objects, as long as you cut off that root reference before the next GC, those million objects contribute nothing to the amount of time the GC takes.

Second, the frequency of a garbage collection is primarily determined by how much memory is allocated in a specific generation. Once that amount passes an internal threshold, a GC will happen for that generation. The threshold continually changes and the GC adapts to your process's behavior. If doing a collection on a particular generation is productive (it promotes many objects), then it will happen more frequently, and the converse is true. Another trigger for GCs is the total available memory on a machine, independent of your application. If available memory drops below a certain threshold, garbage collection may happen more frequently in an attempt to reduce the overall heap size.

From this description, it may feel like garbage collections are out of your control. This could not be farther from the truth! Manipulating GC behavior by controlling your memory allocation patterns is usually possible. It requires understanding of how the GC works, your allocation rate, how well you control object lifetimes, and what configuration options are available to you. Let's take a closer look at those configuration options next.

Detailed Heap Layout

Now that we have seen how it works conceptually, examine these more detailed heap diagrams, drawn from a debugger session via `!eeheap -gc`.

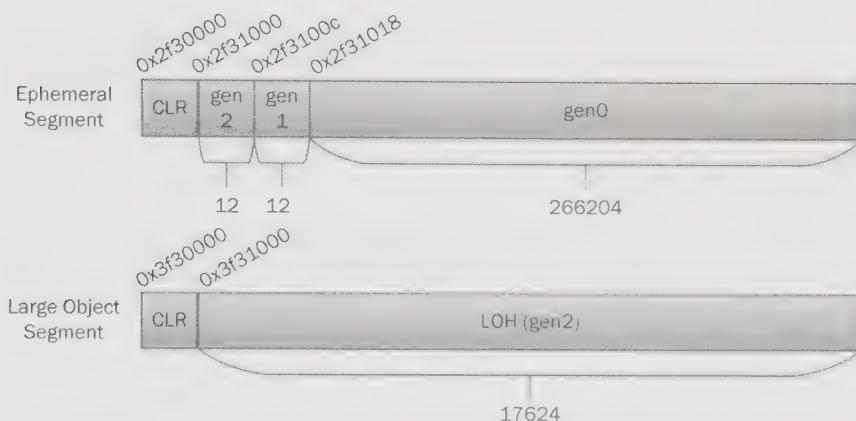


Figure 2.4. Initial heap layout of a sample application with an ephemeral and a large object segment. The total heap size in the figure is about 258KB with about 4KB being used by the CLR directly. Gen 1 and gen 2 are only 12 bytes each.

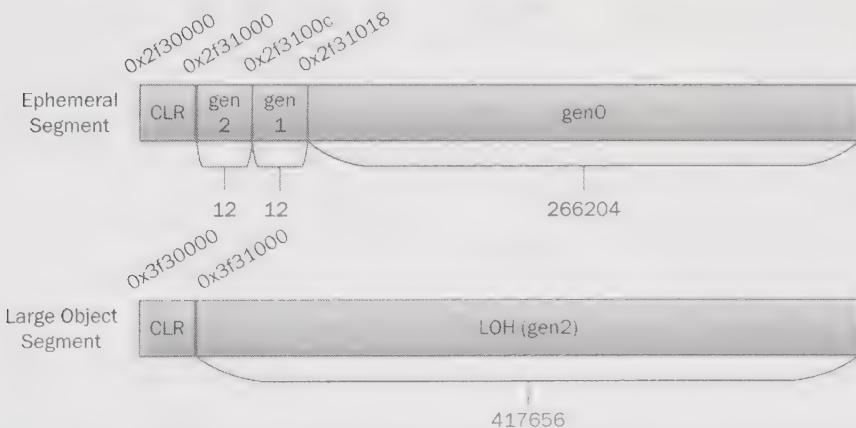


Figure 2.5. After many small allocations and a large array, the ephemeral heap is unchanged – none of the allocations changed any boundaries. However, the LOH expanded from 17KB to over 400KB. Then a GC will happen.

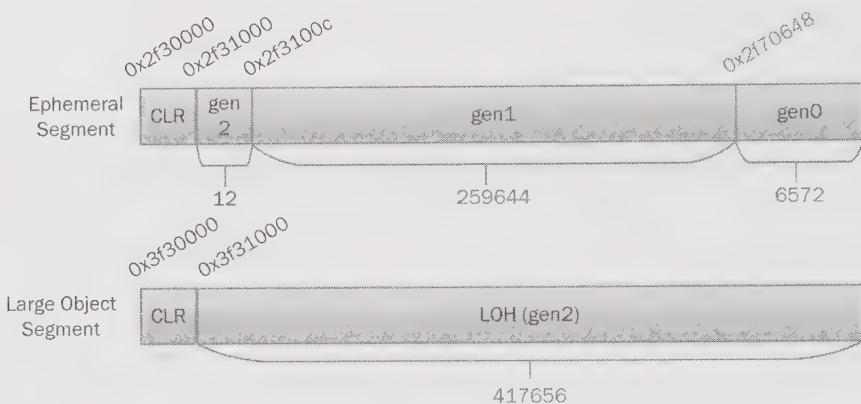


Figure 2.6. After the collection, the large object heap is unchanged, but there are significant modifications to the ephemeral heap. Gen 1 grew significantly, and gen 0 shrunk correspondingly.

Configuration Options

The .NET Framework does not give you very many ways to configure the garbage collector out the box. It is best to think of this as “less rope to hang yourself with.” For the most part, the garbage collector configures and tunes itself based on your hardware configuration, available resources, and application behavior. What few options are provided are for very high-level behaviors, and are mainly determined by the type of program you are developing.

Workstation vs. Server GC

The most important choice you have is whether to use workstation or server garbage collection.

Workstation GC is the default. In this mode, all GCs happen on the same thread that triggered the collection and run at the same priority. For simple apps, especially those that run on interactive workstations where many managed pro-

cesses run, this makes the most sense. For computers with a single processor, this is the only option and trying to configure anything else will not have any effect.

Server GC creates a dedicated thread for each logical processor or core. These threads run at highest priority (THREAD_PRIORITY_HIGHEST), but are always kept in a suspended state until a GC is required. All garbage collections happen on these threads, not the application's threads. After the GC, they sleep again.

In addition, the CLR creates a separate heap for each processor. Within each processor heap, there is a small object heap and a large object heap. From your application's perspective, this is all logically the same heap – your code does not know which heap objects belong to and object references exist between all the heaps (they all share the same virtual address space).

Having multiple heaps gives a couple of advantages:

- Garbage collection happens in parallel. Each GC thread collects one of the heaps. This can make garbage collection significantly faster than in workstation GC.
- In some cases, allocations can happen faster, especially on the large object portion of the heap, where allocations are spread across all the heaps.

There are other internal differences as well such as larger segment sizes, which can mean a longer time between garbage collections.

You configure server GC in the app.config file inside the `<runtime>` element:

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

Should you use workstation or server GC? If your app is running on a multi-processor machine dedicated to just your application, then the choice is clear: server GC. It will provide the lowest latency collection in most situations. However, server GC also means a much higher working set which means you will get closer to physical memory limits. With more objects in memory, garbage collections may start taking longer, eating away at the advantage.

On the other hand, if you need to share the machine with multiple managed processes, the choice is not so clear. Server GC creates many high-priority threads and if multiple apps do that, they can all negatively affect one other with conflicting thread scheduling. In this case, it might be better to use workstation GC.

If you really want to use server GC in multiple applications on the same machine, another option is to affinitize the competing applications to specific processors. The CLR will create heaps only for the processors which are enabled for that application.

Whichever one you pick, most of the tips in this book apply to both types of collection.

Background GC

Background GC changes how the garbage collector performs gen 2 collections by allowing it to occur more often in the background while other threads are executing. Gen 0 and gen 1 collections remain foreground GCs that block all application threads from executing.

Background GC works by having a dedicated thread for garbage collecting generation 2. For server GC there will be an additional thread per logical processor, in addition to the one already created for server GC in the first place. Yes, this means if you use server GC and background GC, you will have two threads per processor dedicated to GC, but this is not particularly concerning. It is not a big deal for processes to have many threads, especially when most of them are doing nothing most of the time. One thread is for foreground GC and runs at highest priority, but it is suspended most of the time. The thread for back-

ground GC runs at a lower priority concurrently with your application's threads and will be suspended when the foreground GC threads become active, so that you do not have competing GC modes occurring simultaneously.

If you are using workstation GC, then background GC is always enabled. Starting with .NET 4.5, it is enabled on server GC by default, but you do have the ability to turn it off.

This configuration will turn off the background GC:

```
<configuration>
  <runtime>
    <gcConcurrent enabled="false"/>
  </runtime>
</configuration>
```

In practice, there should rarely ever be a reason to disable background GC. It will usually cause worse performance and more frequent foreground collections. If you want to prevent these background GC threads from ever taking CPU time from your application, but do not mind a potential increase in full, blocking GC latency or frequency, then you can turn this off. You should measure the impact carefully.

Latency Modes

The garbage collector has a number of latency modes, most of them accessed via the `GCSettings.LatencyMode` property. The mode should rarely be changed, but the options can be useful at times.

`Interactive` is the default GC latency mode when concurrent garbage collection is enabled (which is on by default). This mode allows collections to run in the background.

`Batch` mode disables all concurrent garbage collection and forces collections to occur in a single batch. It is intrusive because it forces your program to stop completely during all GC's. It should not regularly be used, especially in programs with a user interface.

There are two low-latency modes you can use for a limited time. If you have periods of time that require critical performance, you can tell the GC not to perform expensive gen 2 collections.

- `LowLatency`: For workstation GC only, it will suppress gen 2 collections.
- `SustainedLowLatency`: For workstation and server GC, it will suppress full gen 2 collections, but it will allow background gen 2 collections. You must enable background GC for this option to take effect.

Both modes will greatly increase the size of the managed heap because compaction will not occur. If your process uses a lot of memory, you should avoid this feature.

Right before entering one of these modes, it is a good idea to force a last full GC by calling `GC.Collect(2, GCCollectionMode.Forced)`. Once your code leaves this mode, do another GC.

You should never use either of the low-latency modes by default. It is designed for applications that must run without serious interruptions for a long time, but not 100% of the time. A good example is stock trading. During market hours, you do not want full garbage collections happening. When the market closes, you turn this mode off and perform full GCs until the market reopens.

Only turn on a low-latency mode if all of the following criteria apply:

- The latency of a full garbage collection is never acceptable during normal operation.
- The application's memory usage is far lower than available memory. (If you want low-latency mode, then max out your physical memory.)
- Your program can survive long enough until it turns off low-latency mode, restarts itself, or manually performs a full collection.

Finally, starting in .NET 4.6, you can declare regions where garbage collections are disallowed, using the `NoGCRegion` mode. This attempts to put the GC in a mode where it will not allow a GC to happen at all. It cannot be set via this property, however. Instead, you must use the `TryStartNoGCRegion` method.

There are some significant caveats:

- You must specify your total expected memory allocation up front.
- The amount you requested must be at most the size of an ephemeral segment. (See earlier in this chapter for a discussion of segment sizes.)
- The CLR must be able to immediately allocate what you requested for both the ephemeral and the large object heap memory.

There are a number of overloads of `TryStartNoGCRegion`, but the following example demonstrates the one with all of the options:

```
bool success = GC.TryStartNoGCRegion(
    totalSize: 2000000,
    lohSize: 1000000,
    disallowFullBlockingGC: true);

if (success)
{
    try
    {
        // do allocations
    }
    finally
    {
        if (GCSettings.LatencyMode == GCLatencyMode.NoGCRegion)
        {
            GC.EndNoGCRegion();
        }
    }
}
```

The `totalSize` parameter is the total number of bytes that you expect to allocate in the region. The `lohSize` parameter indicates how many of them you expect to be on the large object heap. The difference between `totalSize` and `lohSize` is the amount you expect to allocate on the ephemeral heap, and must be less than or equal to the size of the ephemeral heap (which size is given at the beginning of this chapter). By default, if the memory cannot be allocated by the CLR, it will do a full blocking GC to attempt to free some space. The `disallowFullBlockingGC` parameter can disable this functionality.

You should only call `EndNoGCRegion` if the previous call to `TryStartNoGCRegion` succeeded. You cannot nest calls to `TryStartNoGCRegion`.

If your memory allocations go over the amount you reserved, the guarantee is no longer honored and a garbage collection could happen.

Note

The low-latency or no-GC modes are not absolute guarantees. If the system is running low on memory and the garbage collector has the choice between doing a full collection or throwing an `OutOfMemoryException`, it will perform a full collection regardless of your mode setting.

Alternative latency modes are rarely used and you should think twice about using them because of the potential unintended consequences. If you think it is useful, perform careful measurement to make sure. Tweaking the latency mode may cause other performance problems such as having more ephemeral collections (gen 0 and 1) in an attempt to deal with the lack of full collections. You may just trade one set of problems for another.

Large Objects

By default, arrays are limited to both `UInt32.MaxValue` in number of elements and 2 GB in actual size. Using a configuration option, you can allow larger array sizes, but the maximum number of elements remains the same.

```
<configuration>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

This allows 64-bit processes to have arrays that span more than 2 GB in size. However:

- The maximum number of elements is `UInt32.MaxValue` (4,294,967,295).
- The maximum index of any dimension is 2,147,483,591 for single-byte element arrays, or 2,146,435,071 for other types.
- The maximum size of other objects is unaffected.

Advanced Options

Some GC options must be configured before the process starts because they are required during CLR initialization. In general, these settings will very rarely be necessary and you should strongly consider whether you need them.

These settings are configured via environment variables which are set on the command-line before you launch the process (which will receive a copy of the current environment).

Limit Heap Count

In server GC, there is a heap and at least one thread created for each processor. There may be times when you want to use fewer processors for GC, perhaps in tandem with changing the application's processor affinity mask.

```
// Limit to using the first 16 processors
Process currentProcess = Process.GetCurrentProcess();
long mask = (long)currentProcess.ProcessorAffinity;
```

```
mask &= 0xFFFF;  
currentProcess.ProcessorAffinity = (IntPtr)mask;
```

If the application is launched with processor affinity already applied, then server GC will automatically restrict the number of heaps and threads it creates for garbage collecting.

However, this limits the number of processors the application can use for general work as well. If you want your application to use all of the processors for its own work, but only run GC on a subset of those processors, you need to set the `GCHeapCount` variable, which was introduced to CoreCLR in mid 2016, or .NET Framework 4.7.

```
SET COMPLUS_GCHeapCount=<n>
```

This option is only valid when using server GC. Replace `<n>` with a number less than the number of logical processors in use.

You may want to use this if you need the benefits of server GC, but need to limit the amount of CPU used during GCs. Because server GCs run at a high priority, having a thread per core will stall all other processes on the machine. Usually, this is by design and there is an assumption that a server GC app “owns” the machine, but this option is there if you want to free up some processors. For example, you may have a 64-processor server and you want the parallelism and dedicated, fast GC threads that come with server GC, but 64 heaps may be overkill if you need to be more frugal and ensure other processes do not starve during GCs. In addition, you will lessen the amount of memory overhead if your total memory requirements are more modest.

Disable GC Thread Affinity

In normal circumstances, each server GC thread is affinitized to run on a specific logical processor. This means that during a GC, it is a virtual guarantee that the GC thread will take over the processor as the highest-priority running thread.

With the following setting, you can turn off affinitization, which will allow GC threads to run on any available processor. This will ensure that the server GC process will cooperate better with other processes.

```
SET COMPLUS_GCNoAffinitize=1
```

This setting is designed to work well with COMPLUS_GCHeadCount when you are improving the cooperation between your server GC application with other processes on the machine.

By turning this on, you are explicitly stating that you want more cooperation and less exclusivity. This means there is no chance that this setting improves your application's performance, but it might improve your overall system performance.

Verify Heap

When optimizing code to achieve the highest levels of performance it is unfortunately common to take shortcuts that can lead to bugs such as corrupting program state or even the heap structure itself. Heap corruption in .NET applications is almost always the result of buggy unmanaged code in the same process. However, it is still possible in managed-only applications and can indicate a bug within the CLR itself. When this happens, it can be extremely hard to debug as the crash will not happen at a deterministic place.

You can use the !VerifyHeap command to verify the heap within the debugger.

```
0:006> !VerifyHeap
object 04b05980: bad member 00000066 at 04B05984
Last good object: 04B057E4.
0:006> !do 04B057E4
Name:      System.Int32[]
MethodTable: 62281938
EEClass:    61e09600
Size:      412(0x19c) bytes
```

Array: Rank 1, Number of elements 100, Type Int32
Fields:
None

Also, it can be tricky to deliberately get the heap into a state where problems manifest reliably. The heap can be in an in-between state while a GC is happening so you need to take care to ensure you validate the heap only outside of a GC.

Thankfully, there is an easy way to do this outside of the debugger. You can turn on an option to cause the heap to be verified before and after every GC.

```
SET COMPLUS_HeapVerify=1
```

Turning on heap verification will cause performance to suffer as each GC will now force the heap to be validated, a process which will take longer depending on the size of your heap. If corruption is detected, an exception will be thrown and the process will be terminated.

Performance Tips

Reduce Allocation Rate

This almost goes without saying, but if you reduce the amount of memory you are allocating, you reduce the pressure on the garbage collector to operate. You can also reduce memory fragmentation and CPU usage as well. It can take some creativity to achieve this goal and it might conflict with other design goals.

Critically examine each object and ask yourself:

- Do I really need this object at all?
- Does it have fields that I can get rid of?

- Can I reduce the size of arrays?
- Can I reduce the size of primitives (Int64 to Int32, for example)?
- Are some objects used only in rare circumstances and can therefore be initialized only when needed?
- Can I convert some classes to structs so they live on the stack, or as part of another object, and have no per-instance overhead?
- Am I allocating a lot of memory, to use only a small portion of it?
- Can I get this information in some other way?
- Can I allocate memory up front?

In a server that handled user requests, we found out that one type of common request caused more memory to be allocated than the size of a heap segment. Since the CLR caps the maximum size of segments and gen 0 must exist in a single segment, we were guaranteed a GC on every single request. This is not a good spot to be in because there are few options besides reducing memory allocations.

The Most Important Rule

There is one fundamental rule for high-performance programming with regard to the garbage collector. In fact, the garbage collector was explicitly designed with this idea in mind:

Collect objects in gen 0 or not at all.

Put differently, you want objects to have an extremely short lifetime so that the garbage collector will never touch them at all, or, if you cannot do that, they should go to gen 2 as fast as possible and stay there forever, never to be collected. This means that you maintain a reference to long-lived objects forever. Often, this also means pooling reusable objects, especially anything on the large object heap.

Garbage collections get more expensive in each generation. You want to ensure there are many gen 0/1 collections and very few gen 2 collections. Even with background GC for gen 2, there is still a CPU cost that you would rather not pay: a processor the rest of your program should be using.

Note

You may have heard the myth that you should have 10 gen 0 collections for each gen 1 collection and 10 gen 1 collections for each gen 2 collection. This is not true. Just understand that you want to have lots of fast gen 0 collections and very few of the expensive gen 2 collections.

You want to avoid objects being promoted to gen 1 because those that are will tend to also be promoted to gen 2 in due course. Gen 1 is a sort of buffer before you get to gen 2.

Ideally, every object you allocate goes out of scope by the time the next gen 0 comes around. You can measure how long that interval is and compare it to the duration that data is alive in your application. See the end of the chapter for how to use tools to discover this information.

Obeying this rule requires a fundamental shift in your mindset if you are not used to it. It will inform nearly every aspect of your application, so get used to it early and think about it often.

Reduce Object Lifetime

The shorter an object's lifetime, the less chance it has of being promoted to the next generation when a GC comes along. In general, you should not allocate objects until right before you need them. The exception would be when the cost of object creation is so high it makes sense to create them at an earlier point when it will not interfere with other processing.

On the other side of the object use, you want to make sure that objects go out of scope as soon as possible. For local variables, this can be after the last local usage, even before the end of the method. You can lexically scope it narrower by using the { } brackets, but this will probably not make a practical difference because the compiler will generally recognize when a local object is no longer used anyway. If your code spreads out operations on an object, try to reduce the time between the first and last uses so that the GC can collect the object as early as possible.

Rarely, you may find a need to explicitly `null` out a reference to a temporary object if it is a member or static field on a long-lived object. You would do this only if you want to prevent this object from being promoted by the garbage collector. First, try to change the design to make the reference a local variable where object life time is not as much an issue. If you decide to `null` out a field, this may make the code slightly more complicated because you will have more checks for `null` values scattered around. This can also create a tension between efficiency and always having full state available, particularly for debugging. One option to get around that problem is to convert the object you want to `null` out to another form. For example, serialize an XML document hierarchy to a string, or a temporary state object to a log message that can more efficiently record the state for debugging later. This technique is usually only necessary for large, temporary object graphs that are in fields for convenience purposes.

Another way to manage this balance is to have variable behavior: run your program (or a specific portion of your program, say for a specific request) in a mode that does not `null` out references but keeps them around as long as possible for easier debugging.

Balance Allocations

As described at the beginning of this chapter, the GC works by following object references. In server GC, it does this on multiple threads at once. You want to exploit parallelism as much as possible, and if one thread hits a very long chain of nested objects, the entire garbage collection process will not finish until that long-running thread is complete. In addition, if a particular thread allocates more memory than others, it will trigger a GC more often than if the same allocations were spread across multiple heaps.

Thankfully, there are load-balancing algorithms. For allocations, when the GC detects that heaps are becoming unbalanced, it will start forcing allocations to occur on different heaps. This functionality has existed for the small object heap for many CLR versions, but balancing the large object heap has only happened since version 4.5. On the collection side, cores that run out of collection work can steal work from other heaps.

Problems with unbalanced heaps are less common now with these GC features, but if you suspect too-frequent, or long GC pauses, it may be worth looking at your code for the presence of deep object trees or a thread bias for allocations.

If you do find that a single thread is responsible for most of the allocations, investigate ways to spread this responsibility around. Ensure that you are using Task objects or the thread pool to even out the possibility of different threads handling different requests. Avoid the pattern of a single thread processing a queue of requests and doing the bulk of allocations before handing off the work to other threads to finish processing.

Reduce References Between Objects

Objects that have many references to other objects will take more time for the garbage collector to traverse. A long GC pause time is often an indication of a large, complex object graph.

Another danger is that it becomes much harder to predict object lifetimes if you cannot easily determine all of the possible references to them. Reducing this complexity is a worthy goal just for sane code practices, but it also makes debugging and fixing performance problems easier.

Also, be aware that references between objects of different generations can cause inefficiencies in the garbage collector, specifically references from older objects to newer objects. For example, if an object in generation 2 has a reference to an object in generation 0, then every time a gen 0 GC occurs, a portion of gen 2 objects will also have to be scanned to see if they are still holding onto this reference to a generation 0 object. It is not as expensive as a full GC, but it is still unnecessary work if you can avoid it.

Avoid Pinning

Pinning an object fixes it in place so that the garbage collector cannot move it. Pinning exists so that you can safely pass managed memory references to unmanaged code. It is most commonly used to pass arrays or strings to unmanaged code, but is also used to gain direct **fixed** memory access to data structures or fields. If you are not doing interop with unmanaged code and you do not have any **unsafe** code, then you should not have the need to pin at all. However, even if you avoid explicit pinning in your own code, there are plenty of common APIs that need to do it anyway.

While the pinning operation itself is inexpensive, it throws a bit of a wrench into the garbage collector's operation by increasing the likelihood of fragmentation. The garbage collector tracks those pinned objects so that it can use the free spaces between them, but if you have excessive pinning, it can still cause fragmentation and heap growth.

Pinning can be either explicit or implicit. Explicit pinning is performed with use of a `GCHandle` of type `GCHandleType.Pinned` or the `fixed` keyword and must be inside code marked as `unsafe`. The difference between using `fixed` or a handle is analogous to the difference between `using` and explicitly calling `Dispose`. `fixed` is more convenient, but cannot be used in asynchronous situations, whereas you can pass around a handle and dispose of it in the callback.

Implicit pinning is more common, but can be harder to see and more difficult to remove. The most obvious source of pinning will be any objects passed to unmanaged code via Platform Invoke (P/Invoke). This is not just your own code – managed APIs that you call can, and often do, call native code, which will require pinning.

The CLR will also have pinned objects in its own data structures, but these should normally not be a concern.

Ideally, you should eliminate as much pinning as you can. If you cannot quite do that, follow the same rules for garbage collection: keep lifetime as short as possible. If objects are only pinned briefly then there is less chance for them to affect the next garbage collection. You also want to avoid having very many pinned objects at the same time. Pinning objects located in gen 2 or the LOH is generally fine because these objects are unlikely to move anyway. This can lead to a strategy of either allocating large buffers on the large object heap and giving out portions of them as needed, or allocating small buffers on the small object heap, but before pinning, ensure they are promoted to gen 2. This takes a bit of management on your part, but it can completely avoid the issue of having pinned buffers during a gen 0 GC.

Avoid Finalizers

Never implement a finalizer unless it is required. Finalizers are code, triggered by the garbage collector to cleanup unmanaged resources. They are called from a single thread, one after the other, and only after the garbage collector declares the object dead after a collection. This means that if your class implements a finalizer, you are guaranteeing that it will stay in memory even after the collection that should have killed it. There is also additional bookkeeping to be done on each GC as the finalizer list needs to be continually updated if the object is relocated. All of this combines to decrease overall GC efficiency and ensure that your program will dedicate more CPU resources to cleaning up your object.

Not only that, but an object with a finalizer is slower to allocate. Instead of the “fast path” allocator, it must do extra bookkeeping to ensure the GC tracks the object for its lifetime.

If you do implement a finalizer, you must also implement the `IDisposable` interface to enable explicit cleanup, and call `GC.SuppressFinalize(this)` in the `Dispose` method to remove the object from the finalization queue. As long as you call `Dispose` before the next collection, it will clean up the object prop-

erly without the need for the finalizer to run. The following example correctly demonstrates this pattern. Note that you can (and often should) implement the Dispose pattern without implementing a finalizer.

```
class Foo : IDisposable
{
    private bool disposed = false;
    private IntPtr handle;
    private IDisposable managedResource;

    ~Foo() // Finalizer
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (this.disposed)
        {
            return;
        }
        if (disposing)
        {
            // Not safe to do this from finalizer
            this.managedResource.Dispose();
        }

        // Cleanup unmanaged resources that are safe to
        // do so in a finalizer
        UnsafeClose(this.handle);

        // If the base class is IDisposable object
        // make sure you call base.Dispose(disposing);
        this.disposed = true;
    }
}
```

All cleanup logic is centralized in the `Dispose(bool)` method. Everything else just calls it. The `disposing` variable indicates whether a developer explicitly called `Dispose`. If they did, then it is safe to `Dispose` of all resources. However, if this method is called via the finalizer, then there is no guarantee any referenced objects are still valid, so only those unmanaged resources explicitly owned by this object can be safely cleaned up in this method. In the context of a finalizer, very few assumptions can be made about the state of objects referenced by this object. The code must be simple and touch only memory guaranteed to belong only to this object and still be valid. Typically this means that you should not access any other finalizable object, or any other disposable object (unless you can guarantee its validity).

Only mark the `protected` version of `Dispose` virtual and allow it to be overridden by child types. The `disposed` field tracks whether the object has already been disposed, allowing the `Dispose` method to be called more than once.

`Dispose` methods and finalizers should never throw exceptions. Should an exception occur during a finalizer's execution, then the process will terminate. Finalizers should also be careful doing any kind of I/O, even as simple as logging.

Properly implementing this pattern is important to ensure that it works correctly with polymorphic types. You will have to exercise judgment on whether to implement finalizers on base types that themselves do not have unmanaged resources, but may have derived types that do have such resources. It may be required in some cases to take the performance hit for correctness, but this should be avoided if at all possible.

Any type that contains instances of other `IDisposable` types must itself implement `IDisposable`. In this way, `IDisposable` has a way of spreading through your data structures. Properly implemented, it should be easy to dispose of all the resources merely by calling the root `IDisposable`'s `Dispose` method.

Note

You may have heard that finalizers are guaranteed to run. This is generally true, but not absolutely so. If a program is force-terminated then no more code runs and the process dies imme-

dately. The finalizer thread is triggered by a garbage collection, so if there are no garbage collections, finalizers will not run.

There is also a time limit to how long all of the finalizers are given on process shutdown. If your finalizer is at the end of the list, it may be skipped. Moreover, because finalizers execute sequentially, if another finalizer has an infinite loop bug in it, then no finalizers after it will ever run. This can lead to memory leaks. For all these reasons, you should not rely on finalizers to clean up state external to your process.

Avoid Large Object Allocations

Not all allocations go to the same heap. Objects over a certain size will go to the large object heap and immediately be in gen 2. The boundary for large object allocations was set at 85,000 bytes by doing a statistical analysis of programs of the day. Any object of that size or greater is judged to be “large” and it goes on a separate heap.

You want to avoid allocations on the large object heap as much as possible. Not only is collecting garbage from this heap more expensive, it is more likely to fragment, causing unbounded memory increases over time. Continuous allocations to the large object heap send a strong signal to the garbage collector to do continuous garbage collections—not a good place to be in.

To avoid these problems, you need to strictly control what your program allocates on the large object heap. What does go there should last for the lifetime of your program and be reused as necessary in a pooling scheme.

The large object heap does not automatically compact, but you may tell it to do so programmatically starting with .NET 4.5.1. However, you should use this only as a last resort, as it will cause a very long pause. Before explaining how to do that, the next few sections will explain how to avoid getting into that situation in the first place.

Avoid Copying Buffers

You should usually avoid copying data whenever you can. For example, suppose you have read file data into a `MemoryStream` (preferably a pooled one if you need large buffers). Once you have that memory allocated, treat it as read-only and every component that needs to access it will read from the same copy of the data.

A common requirement, then, is to refer to sub-ranges of a buffer, array, or memory range. .NET provides two ways to accomplish this at present.

The first option, available only for arrays, is the `ArraySegment<T>` struct to represent just a portion of the underlying array. This `ArraySegment` can be passed around to APIs independent of the original stream, and you can even attach a new `MemoryStream` to just that segment. Throughout all of this, no copy of the data has been made.

```
var memoryStream = new MemoryStream(2048);
var segment = new ArraySegment<byte>(memoryStream.GetBuffer(),
                                         100,
                                         1024);
...
var blockStream = new MemoryStream(segment.Array,
                                    segment.Offset,
                                    segment.Count);
```

The biggest problem with copying memory is not the CPU necessarily, but the GC. If you find yourself needing to copy a buffer, then try to copy it into another pooled or existing buffer to avoid any new memory allocations.

A newer option for representing pieces of existing buffers is the `Span<T>` struct. `Span<T>` is still in a pre-release phase at the time of this writing, but it will likely become finalized with the release of C# 7.2 or future upgrades to the runtime. To use this library, you will need to consume the `System.Memory` NuGet package and use Visual Studio 2017.

Span<T> is like an array in the sense that it represents a contiguous block of memory, but it has the distinction of being able to wrap managed memory, unmanaged memory, and stack memory with the same abstraction. For unmanaged memory, you can think of it as a smart wrapper that does pointer arithmetic for you.

The following examples of Span<T> come from the Span project in the accompanying sample code.

The first example creates a standard byte array on the managed heap and creates a span from a sub-portion of that array. (It could just as easily have spanned the entire array.)

```
{  
    ...  
    byte[] array = new byte[] {0, 1, 2, 3};  
    Span<byte> byteSpan = new Span<byte>(array, 1, 2);  
    PrintSpan(byteSpan);  
    ...  
}  
  
private static void PrintSpan<T>(Span<T> span)  
{  
    for (int i = 0; i < span.Length; i++)  
    {  
        ref T val = ref span[i];  
        Console.Write(val);  
        if (i < span.Length - 1) { Console.Write(", "); }  
    }  
    Console.WriteLine();  
}
```

This produces the following output:

1, 2

This example uses a `Span<T>` to wrap a stack-allocated array:

```
unsafe
{
    int* stackMem = stackalloc int[4];
    Span<int> intSpan = new Span<int>(stackMem, 4);
    for (int i=0;i<intSpan.Length;i++)
    {
        intSpan[i] = 13 + i;
    }
    PrintSpan(intSpan);
}
```

As you can see, it uses the exact same semantic to wrap this array, and the same helper method can be used to print the values. Its output is:

13, 14, 15, 16

The next example is slightly more complex. When you allocate from the native heap, you must specify the number of bytes you are allocating and when you wrap unmanaged memory in a `Span<T>`, you are assigning types to that memory, so the length of the span is specified in the count of objects, not length of bytes. This example accounts for that by multiplying the size of the objects we want by the count before we allocate.

```
unsafe
{
    const int ObjectCount = 4;
    int memSize = sizeof(int) * ObjectCount;
    IntPtr hNative = Marshal.AllocHGlobal(memSize);
    Span<int> unmanagedSpan = new Span<int>(hNative.ToPointer(),
                                                ObjectCount);
    for (int i = 0; i < unmanagedSpan.Length; i++)
    {
        unmanagedSpan[i] = 100 + i;
    }
    PrintSpan(unmanagedSpan);
    Marshal.FreeHGlobal(hNative);
}
```

The output is:

```
100, 101, 102, 103
```

The final example makes use of one of the extension methods included in the library to convert a string into a `ReadOnlySpan<char>`. Unfortunately, there is no relationship between `Span<T>` and `ReadOnlySpan<T>` because `Span<T>` utilizes `ref`-return semantics to avoid copying values. That means, we have to have a separate utility method to print the values.

```
{  
    ...  
    ReadOnlySpan<char> subString =  
        "NonAllocatingSubstring".AsSpan().Slice(13);  
    PrintSpan(subString);  
    ...  
  
    private static void PrintSpan<T>(ReadOnlySpan<T> span)  
    {  
        for (int i = 0; i < span.Length; i++)  
        {  
            T val = span[i];  
            Console.Write(val);  
            if (i < span.Length - 1) { Console.Write(", "); }  
        }  
        Console.WriteLine();  
    }  
}
```

The output of this code is:

```
S, u, b, s, t, r, i, n, g
```

There are also utility methods to convert from arrays and `ArraySegment` structs to `Span<T>` structs.

Pool Long-Lived and Large Objects

Remember the cardinal rule from earlier: Objects live very briefly or forever. They should either go away in gen 0 collections or last forever in gen 2. Some objects are essentially static—they are created and last the lifetime of the program naturally. Other objects do not obviously need to last forever, but their natural lifetime in the context of your program ensures they will live longer than the period of a gen 0 (and maybe gen 1) garbage collection. These types of objects are candidates for pooling. Another strong candidate for pooling is any object that you allocate on the large object heap, typically collections.

There is no single way to pool and there is no standard pooling API you can rely on. It really is up to you to develop a way that works for your application and the specific objects you need to pool.

One way to think about poolable objects is that you are turning a normally managed resource (memory) into something that you have to manage explicitly. .NET already has a pattern for dealing with finite managed resources: the `IDisposable` pattern. See earlier in this chapter for the proper implementation of this pattern. A reasonable design is to derive a new type and have it implement `IDisposable`, where the `Dispose` method puts the pooled object back in the pool. This will be a strong clue to users of that type that they need to treat this resource specially.

Implementing a good pooling strategy is not trivial and can depend entirely on how your program needs to use it, as well as what types of objects need to be pooled. Here is code that shows one example of a simple pooling class to give you some idea of what is involved. This code is from the `PooledObjects` sample program.

```
interface IPoolableObject : IDisposable
{
    int Size { get; }
    void Reset();
    void SetPoolManager(PoolManager poolManager);
}

class PoolManager
{
```

```
private class Pool
{
    public int PooledSize { get; set; }
    public int Count { get { return this.Stack.Count; } }
    public Stack<IPoolableObject> Stack { get; private set; }
    public Pool()
    {
        this.Stack = new Stack<IPoolableObject>();
    }

}
const int MaxSizePerType = 10 * (1 << 10); // 10 MB

Dictionary<Type, Pool> pools =
    new Dictionary<Type, Pool>();

public int TotalCount
{
    get
    {
        int sum = 0;
        foreach (var pool in this.pools.Values)
        {
            sum += pool.Count;
        }
        return sum;
    }
}

public T GetObject<T>()
    where T : class, IPoolableObject, new()
{
    Pool pool;
    T valueToReturn = null;
    if (pools.TryGetValue(typeof(T), out pool))
    {
        if (pool.Stack.Count > 0)
        {
            valueToReturn = pool.Stack.Pop() as T;
        }
    }
    if (valueToReturn == null)
    {
```

```
        valueToReturn = new T();
    }
    valueToReturn.SetPoolManager(this);
    pool.PooledSize -= valueToReturn.Size;

    return valueToReturn;
}

public void ReturnObject<T>(T value)
    where T : class, IPoolableObject, new()
{
    Pool pool;
    if (!pools.TryGetValue(typeof(T), out pool))
    {
        pool = new Pool();
        pools[typeof(T)] = pool;
    }

    if (value.Size + pool.PooledSize <= MaxSizePerType)
    {
        pool.PooledSize += value.Size;
        value.Reset();
        pool.Stack.Push(value);
    }
}

class MyObject : IPoolableObject
{
    private PoolManager poolManager;
    public byte[] Data { get; set; }
    public int UsableLength { get; set; }

    public int Size
    {
        get { return Data != null ? Data.Length : 0; }
    }

    void IPoolableObject.Reset()
    {
        UsableLength = 0;
    }
}
```

```
void IPoolableObject.SetPoolManager(  
    PoolManager poolManager)  
{  
    this.poolManager = poolManager;  
}  
  
public void Dispose()  
{  
    this.poolManager.ReturnObject(this);  
}  
}
```

It may seem a burden to force pooled objects to implement a custom interface, but apart from convenience, this highlights a very important fact: In order to use pooling and reuse objects, you must be able to fully understand and control them. Your code must reset them to a known, safe state every time they go back into the pool. This means you should not naively pool 3rd-party objects directly. By implementing your own objects with a custom interface, you are providing a very strong signal that the objects are special. You should especially be wary of pooling objects from the .NET Framework.

It is particularly tricky pooling collections because of their nature – you do not want to destroy the actual data storage (that is the whole point of pooling, after all), but you must be able to signify an empty collection with available space. Thankfully, most collection types implement both `Length` and `Capacity` properties that make this distinction. Given the dangers of pooling the existing .NET collection types, it is better if you implement your own collection types using the standard collection interfaces such as `IList<T>`, `ICollection<T>`, and others. See Chapter 6 for general guidance on creating your own collection types.

An additional strategy is to have your poolable types implement a finalizer as a safety mechanism. If the finalizer runs, it means that `Dispose` was never called, which is a bug. You can choose to write something to the log, crash, or otherwise signal the problem. You must be very careful with this signaling, though, because touching memory that has been invalidated by the GC will cause a crash or hang.

Remember that a pool that never dumps objects is indistinguishable from a memory leak. Your pool should have a bounded size (in either bytes or number of objects), and once that has been exceeded, it should drop objects for the GC to clean up. Ideally, your pool is large enough to handle normal operations without dropping anything and the GC is only needed after brief spikes of unusual activity. Depending on the size and number of objects contained in your pool, dropping them may lead to long, full GCs. It is important to make sure your pool is tunable for your situation.

I do not usually run to pooling as a default solution. As a general-purpose mechanism, it is clunky and error-prone. However, you may find that your application will benefit from pooling of just a few types.

Case Study: RecyclableMemoryStream

I once worked on an application that managed federation to thousands of back-end network resources per second. Most of its work was reading bytes off the network or writing to it. Nearly 90% of all allocated memory was going towards `MemoryStream` objects that were being allocated and resized all over the place: string encoding, marshaling, unmarshaling, temporary buffers, and more. As a result, we were spending a phenomenal amount of time just doing GC – nearly 25% of all CPU time! Doing memory and CPU profiling quickly revealed the need for a better way to handle bytes than `MemoryStream`.

This section will discuss the design and some implementation details of a pooled `MemoryStream` class, called `RecyclableMemoryStream`. You can download the code at <https://github.com/Microsoft/Microsoft.IO.RecyclableMemoryStream> or use it directly from Visual Studio with a NuGet package.

Our requirements for the replacement were:

- Completely eliminate allocations to the large object heap
- Spend less time in GC, especially with gen 2 GCs
- Avoid memory leaks by bounding the pool size

- Avoid memory fragmentation
- Provide easy debuggability
- Instrument for metrics and logging
- Obey `Dispose` semantics
- Be a drop-in replacement for `MemoryStream`, as much as possible
- Have thread safety

These requirements were all met and led to the following list of features and implementation details:

- Instead of pooling the streams themselves, the underlying buffers are pooled. This allows us to better instrument the streams and detect aberrant usage patterns such as stream reuse and leaking (especially important in pooling scenarios), as well as simpler reuse of the arrays. It also helps prevent fragmentation by using equal buffer sizes.
- The streams are an abstraction on top of chained buffers, to appear as a single large buffer.
- While streams themselves are not thread-safe, the acts of allocating and freeing pooled streams are thread-safe.
- Each stream has an identifying tag which can help you debug incorrect pool usage.
- Includes the ability to track the allocation stack of the stream to help you debug pool leaks.
- Allows flexible and configurable pool limits that restrict memory usage while handling spikes in usage.
- Surfaces detailed events and metrics to track usage over time.

The devil is in the details, as the saying goes, so let's dive into some of the implementation.

Before you can allocate a `RecyclableMemoryStream`, you must create the pool manager, a `RecyclableMemoryStreamManager` object. This is the class that actually manages the buffer pools and tracks resource usage. Think of it like a miniature heap inside the CLR's heap. On this class, you set all of your configuration options, like default buffer sizes, maximum size of the heap, and more. There is typically one manager object per process and it lives for the lifetime of the process. However, if you have wildly different usage scenarios, there is no problem using multiple `RecyclableMemoryStreamManager` objects.

The `RecyclableMemoryStreamManager` maintains two categories of buffers: the Small Pool and the Large Pool. The Small Pool is made of lots of equal-sized buffers. The “Small” in Small Pool refers to the size of the individual buffer, not the size of the pool. The buffers in the Small Pool are called blocks (because they are combined to form the longer stream). The Large Pool contains larger buffers, but far fewer of them, and is designed to be used less frequently (only when `GetBuffer` is called). Both pools use uniform buffer sizes to reduce the likelihood of heap fragmentation.

Using this library is easy:

```
var sourceBuffer = new byte[]{0,1,2,3,4,5,6,7};  
var manager = new RecyclableMemoryStreamManager();  
using (var stream = manager.GetStream("Test"))  
{  
    stream.Write(sourceBuffer, 0, sourceBuffer.Length);  
}
```

This code creates a `RecyclableMemoryStreamManager` with default settings, grabs a stream, writes some bytes to it, and then returns the stream's blocks to the pool with the `Dispose` call. This example passes the tag “Test” to the stream's constructor. This tag is not unique per-stream, but serves to identify the location in code where it is allocated, which can help in debugging. It is not required to use tags, but they are useful. Internally, each stream is also assigned a unique GUID that does serve to uniquely identify the stream, which can be useful when tracing concurrent usage of multiple streams.

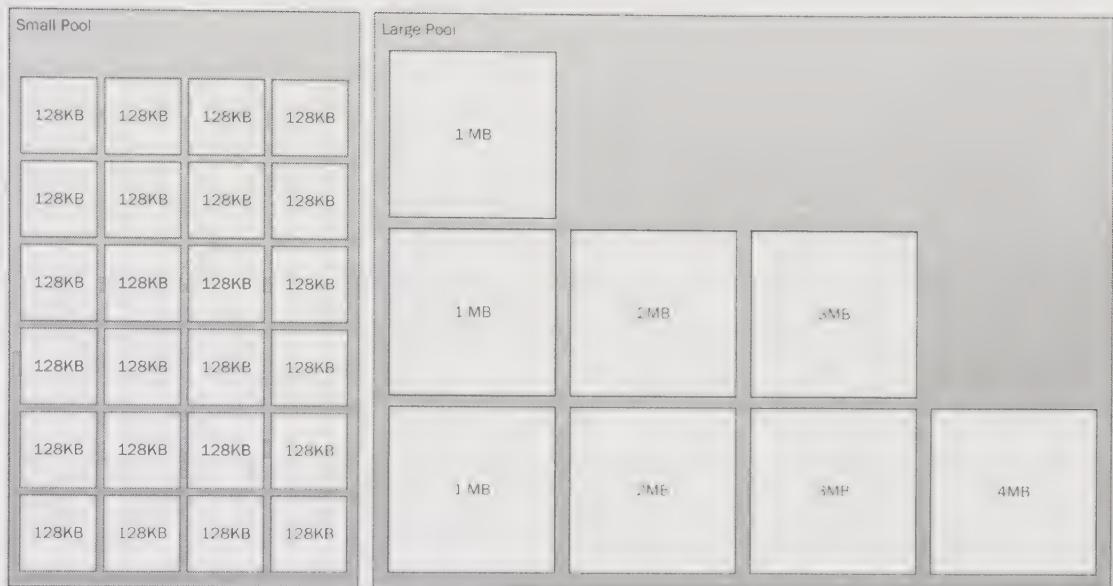


Figure 2.7. The pools in `RecyclableMemoryStreamManager`, with a block size of 128KB, a large buffer multiple of 1MB, and a maximum buffer size of 4 MB.

Internally, the `RecyclableMemoryStream` will grab a block from the manager. As more data is written to the stream, more blocks are chained together and the stream's APIs will make this look like a single contiguous block of memory. As the length of the stream grows, the total memory usage only grows by the block size (and that is assuming the blocks were not already pooled). This is in contrast to `MemoryStream`'s implementation, which doubles the stream's capacity as it grows, leading to potentially massive memory waste, which is fine on a small scale, but not on a massive scale.

As long as just `Read` and `Write` methods are used, only blocks will be used. However, sometimes it is necessary to get a single contiguous buffer. For this, there is the `GetBuffer` API, inherited from `MemoryStream`. When `GetBuffer` is called, a contiguous block must be returned. If there is only one block in use, then a reference to it is returned. If multiple blocks are used, then the Large Pool is used to satisfy the request, and bytes are copied from the blocks to the larger buffer. If the buffer requested is larger than the maximum buffer size of the pool, then a memory allocation occurs to satisfy the request.

It is worthwhile noting that the buffer returned is at least as large as the data contained in it – it may in fact be much larger. You must use the stream’s `Length` property to determine how much data is actually in it. Naive users of the library sometimes ignore this and write huge buffers to the network or to files. After converting the stream to a buffer, with an associated data length, it may be useful to wrap them in an `ArraySegment<byte>` struct.

The `ToByteArray` method is much less useful in a pooling scenario. It is required to return an array of exactly the right size, which means that an allocation (possibly on the large object heap) will occur, as well as a memory copy. Because of these inefficiencies, `ToByteArray` should just be completely avoided.

I encourage you to study the code at the link provided earlier because it will be beneficial to understanding how the library attempts to avoid allocations while balancing the need for other requirements.

Once we implemented this library in production code, we saw allocation on the large object heap drop 99%. Worrying about expensive gen 2 collections became a thing of the past. The time spent in garbage collection dropped from 25% to less than 1%.

Reduce Large Object Heap Fragmentation

If you cannot completely avoid large object heap allocations, then you want to do your best to avoid fragmentation.

The large object heap can grow indefinitely if you are not careful, but it is mitigated by the free list. To take advantage of this free list, you want to increase the likelihood that memory allocations can be satisfied from holes in the heap.

One way to do this is to ensure that all allocations on the LOH are of uniform size, or at least multiples of some standard size. For example, a common need for LOH allocations is for buffer pools. Rather than have a hodge-podge of buffer sizes, ensure that they are all the same size, or in multiples of some well-known number such as one megabyte. This way, if one of the buffers does need to get garbage collected, there is a high likelihood that the next buffer allocation can fill its spot rather than going to the end of the heap.

Force Full GCs in Some Circumstances

In nearly all cases, you should not force collections to happen outside of their normal schedule as determined by the GC itself. Doing so disrupts the automatic tuning the garbage collector performs and may lead to worse behavior overall. However, there are some considerations in a high-performance system that may cause you to reconsider this advice in very specific situations.

In general, it may be beneficial to force a GC to occur during a more optimal time to avoid a GC occurring during a worse time later on. Note that we are only talking about the expensive, ideally rare, full GCs. Gen 0 and gen 1 GCs can and should happen frequently to avoid building up a too-large gen 0 size.

Some situations may merit a forced collection:

1. You are using low-latency GC mode. In this mode, heap size can grow and you will need to determine appropriate points to perform a full collection. See the section earlier in this chapter about low-latency GC.
2. You have a natural downtime, or off-peak hours, in the application's schedule. This may mean you are using a low-latency GC mode, but is not required.
3. You occasionally make a large number of allocations that will live for a long time (forever, ideally). It makes sense to get these objects into gen 2 as quickly as possible. If these objects replace other objects that will now become garbage, you can just get rid of them immediately with a forced collection.
4. You need to compact the large object heap because of fragmentation. See the section about large object heap compaction.

Situations 1 through 3 are all about avoiding full GC's during specific times by forcing them at other times. Situation 4 is about reducing your overall heap size if you have significant fragmentation on the LOH. If your scenario does not fit into one of those categories, you should not consider this a useful option.

To perform a full collection, call the `GC.Collect` method with the generation of the collection you want it to perform. Optionally, you can specify a value of the `GCCollectionMode` enumeration argument to tell the GC to decide for itself whether to do the collection. There are three possible values:

- **Default:** Currently, `Forced`.
- **Forced:** Tells the garbage collector to start the collection immediately.
- **Optimized:** Allows the garbage collector to decide if now is a good time to run.

```
GC.Collect(2);
// equivalent to:
GC.Collect(2, GCCollectionMode.Forced);
```

This exact situation existed on a server that took user queries. Every few hours we needed to reload over a gigabyte of data, replacing the existing data. Since this was an expensive operation and we were already reducing the number of requests the machine was receiving, we also forced two full GCs after the reload happened. This removed the old data and ensured that everything allocated in gen 0 either got collected or made it to gen 2 where it belonged. Then, once we resumed a full query load, there would not be a huge, full GC to affect the first queries.

Compact the Large Object Heap On-Demand

Even if you do pooling, it is still possible that there are allocations you cannot control and the large object heap will become fragmented over time. Starting in .NET 4.5.1, you can tell the GC to compact the large object heap on the next full collection.

```
GCSettings.LargeObjectHeapCompactionMode =  
    GCLargeObjectHeapCompactionMode.CompactOnce;
```

Depending on the size of the large object heap, this can be a slow operation, up to multiple seconds. You may want to put your program in a state where it stops doing real work and force an immediate collection with the `GC.Collect` method.

This setting only affects the next full GC that happens. Once the next full collection occurs, `GCSettings.LargeObjectHeapCompactionMode` resets automatically to `GCLargeObjectHeapCompactionMode.Default`.

Because of the expense of this operation, I recommend you reduce the number of large object heap allocations to as little as possible and pool those that you do make. This will significantly reduce the need for compaction. View this feature as a last resort and only if fragmentation and very large heap sizes are an issue.

Get Notified of Collections Before They Happen

If your application absolutely should not be impacted by gen 2 collections, then you can tell the GC to notify you when a full GC is approaching. This will give you a chance to stop processing temporarily, perhaps by shunting requests off the machine, or otherwise putting the application into a more favorable state.

It may seem like this notification mechanism is the answer to all GC woes, but I recommend extreme caution. You should only implement this after you have optimized as much as you can in other areas. You can only take advantage of GC notifications if all of the following statements are true:

1. A full GC is so expensive that you cannot afford to endure a single one during normal processing.
2. You are able to turn off processing for the application completely. (Perhaps other computers or processes can do the work meanwhile.)

3. You can turn off processing quickly (so you do not waste more time stopping processing than actually performing the GC).
4. Gen 2 collections happen rarely enough to make this worth it.

Gen 2 collections will happen rarely only if you have large object allocations minimized and little promotion beyond gen 0, so it will still take a fair amount of work to get to the point where you can reliably take advantage of GC notifications.

Unfortunately, because of the imprecise nature of GC triggering, you can only specify the pre-trigger time in an approximate way with a number in the range 1-99. With a number that is very low, you will be notified much closer to when the GC will happen, but you risk having the GC occur before you can react to it. With a number that is too high, the GC may be quite far away and you will get a notification far too frequently, which is quite inefficient. It all depends on your allocation rate and overall memory load. Note that you specify two numbers: one for the gen 2 threshold and one for the large object heap threshold. As with other features, this notification is a best effort by the garbage collector. The garbage collector never guarantees you can avoid doing a collection.

To use this mechanism, follow these general steps:

1. Call the `GC.RegisterForFullGCNotification` method with the two threshold values.
2. Poll the GC with the `GC.WaitForFullGCApproach` method. This can wait forever or accept a timeout value.
3. If the `WaitForFullGCApproach` method returns `Success`, put your program in a state acceptable for a full GC (e.g., turn off requests to the machine).
4. Force a full collection yourself by calling the `GC.Collect` method.
5. Call `GC.WaitForFullGCComplete` (again with an optional timeout value) to wait for the full GC to complete before continuing.

6. Turn requests back on.
7. When you no longer want to receive notifications of full GCs, call the `GC.CancelFullGCNotification` method.

Because this requires a polling mechanism, you will need to run a thread that can do this check periodically. Many applications already have some sort of “housekeeping” thread that performs various actions on a schedule. This may be an appropriate task, or you can create a separate dedicated thread.

Here is a full example from the `GCNotification` sample project demonstrating this behavior in a simple test application that allocates memory continuously. See the accompanying source code project to test this.

```
class Program
{
    static void Main(string[] args)
    {
        const int ArrSize = 1024;
        var arrays = new List<byte[]>();

        GC.RegisterForFullGCNotification(25, 25);

        // Start a separate thread to wait for GC notifications
        Task.Run(()=>WaitForGCThread(null));

        Console.WriteLine("Press any key to exit");
        while (!Console.KeyAvailable)
        {
            try
            {
                arrays.Add(new byte[ArrSize]);
            }
            catch (OutOfMemoryException)
            {
                Console.WriteLine("OutOfMemoryException!");
                arrays.Clear();
            }
        }

        GC.CancelFullGCNotification();
    }
}
```

```
}

private static void WaitForGCThread(object arg)
{
    const int MaxWaitMs = 10000;
    while (true)
    {
        // There is also an overload of WaitForFullGCApproach
        // that waits indefinitely
        GCNotificationStatus status =
            GC.WaitForFullGCApproach(MaxWaitMs);
        bool didCollect = false;
        switch (status)
        {
            case GCNotificationStatus.Succeeded:
                Console.WriteLine("GC approaching!");
                Console.WriteLine(
                    "-- redirect processing to another machine -- ");
                didCollect = true;
                GC.Collect();
                break;
            case GCNotificationStatus.Canceled:
                Console.WriteLine("GC Notification was canceled");
                break;
            case GCNotificationStatus.Timeout:
                Console.WriteLine("GC notification timed out");
                break;
        }

        if (didCollect)
        {
            do
            {
                status = GC.WaitForFullGCComplete(MaxWaitMs);
                switch (status)
                {
                    case GCNotificationStatus.Succeeded:
                        Console.WriteLine("GC completed");
                        Console.WriteLine(
                            "-- accept processing on this machine again --");
                        break;
                    case GCNotificationStatus.Canceled:
                        Console.WriteLine(
```

```
        "GC Notification was canceled");
    break;
case GCNotificationStatus.Timeout:
    Console.WriteLine(
        "GC completion notification timed out");
    break;
}
// Looping isn't necessary, but it is useful if you want
// to check other state before waiting again.
} while (status == GCNotificationStatus.Timeout);
}
}
}
```

Another possible reason is to compact the large object heap, but you could trigger this based on memory usage instead, which may be more appropriate.

Use Weak References For Caching

Weak references are references to an object that still allow the garbage collector to clean up the object. They are in contrast to the default strong references, which prevent collection completely (for that object). They are mostly useful for caching expensive objects that you would like to keep around, but are willing to let go if there is enough memory pressure. Weak references are a core CLR concept that are exposed through a couple of .NET classes:

- `WeakReference`
- `WeakReference<T>`

You should ignore the first one in favor of the generic version that was introduced in .NET 4.5. The non-generic version has API weaknesses that are resolved in the newer version, and I will only discuss the generic version here.

An example of a simple usage:

```
// The underlying Foo object can be garbage collected at any time!
WeakReference<Foo> weakRef = new WeakReference(new Foo());
...
// Create a strong reference to the object,
// now no longer eligible for GC
Foo myFoo;
if (weakRef.TryGetTarget(out myFoo))
{
    ...
}
```

Note that the reference to the `WeakReference<T>` object itself is strong, which means that it will not be collected out from under you—it is only the underlying target object that is weakly referenced. If you are memory-conscious enough to use `WeakReference<T>` then you might rightly be leery of continually allocating new `WeakReference<T>` objects. Thankfully, you can reuse these wrapper objects by using the `SetTarget` method to replace the underlying value as needed.

You can still have other references, both strong and weak, to the same object. Collection will only happen if the *only* references to it are weak (or non-existent).

Most applications do not need to use weak references at all, but there are some criteria that may indicate good usage:

- Memory use needs to be tightly restricted (such as on mobile devices).
- Object life time is highly variable. If your object lifetime is predictable, you can just use strong references and control their life time directly.
- Objects are large, but easy to create. Weak references are ideal for objects that would be nice to have around, but if they are not, you can easily regenerate them or do without. (Note that this also implies that the object size should be significant compared to the overhead of using the additional `WeakReference<T>` objects in the first place.)
- You need secondary indexes for objects. See the example below.

Following are two examples of using `WeakReference<T>` for efficient caching.

Example: HybridCache

A good way to use `WeakReference<T>` is as part of a cache. Objects start out held through a strong reference, but after enough time of not being used (or some other criteria of your choosing), they can be demoted to being held by weak references, which may eventually disappear through garbage collection.

This example shows a simple cache that internally manages two levels of caches.

```
public class HybridCache<TKey, TValue> where TValue : class
{
    class ValueContainer<T>
    {
        public T value;
        public long additionTime;
        public long demoteTime;
    }

    private readonly TimeSpan maxAgeBeforeDemotion;

    // Values live here until they hit their maximum age
    private readonly ConcurrentDictionary<TKey,
                                         ValueContainer<TValue>>
    strongReferences =
        new ConcurrentDictionary<TKey, ValueContainer<TValue>>();

    // Values are moved here after they hit their maximum age
    private readonly ConcurrentDictionary<
        TKey,
        WeakReference<ValueContainer<TValue>>>
    weakReferences =
        new ConcurrentDictionary<
            TKey,
            WeakReference<ValueContainer<TValue>>>();

    public int Count
    {
        get
        {
            return this.strongReferences.Count;
        }
    }
}
```

```
public int WeakCount
{
    get
    {
        return this.weakReferences.Count;
    }
}

public HybridCache(TimeSpan maxAgeBeforeDemotion)
{
    this.maxAgeBeforeDemotion = maxAgeBeforeDemotion;
}

public void Add(TKey key, TValue value)
{
    RemoveFromWeak(key);
    var container = new ValueContainer<TValue>();
    container.value = value;
    container.additionTime = Stopwatch.GetTimestamp();
    container.demoteTime = 0;
    this.strongReferences.AddOrUpdate(
        key,
        container,
        (k, existingValue) => container);
}

private void RemoveFromWeak(TKey key)
{
    WeakReference<ValueContainer<TValue>> oldValue;
    weakReferences.TryRemove(key, out oldValue);
}

public bool TryGetValue(TKey key, out TValue value)
{
    value = null;
    ValueContainer<TValue> container;
    if (this.strongReferences.TryGetValue(key, out container))
    {
        AttemptDemotion(key, container);
        value = container.value;
        return true;
    }
}
```

```
WeakReference<ValueContainer< TValue >> weakRef;
if (this.weakReferences.TryGetValue(key, out weakRef))
{
    if (weakRef.TryGetTarget(out container))
    {
        value = container.value;
        return true;
    }
    else
    {
        RemoveFromWeak(key);
    }
}
return false;
}

/// <summary>
/// Call this method periodically from another thread.
/// </summary>
public void DemoteOldObjects()
{
    var demotionList =
        new List<KeyValuePair< TKey,
                               ValueContainer< TValue >>>();

    long now = Stopwatch.GetTimestamp();

    foreach (var kvp in this.strongReferences)
    {
        var age = CalculateTimeSpan(kvp.Value.additionTime,
                                      now);
        if (age > this.maxAgeBeforeDemotion)
        {
            demotionList.Add(kvp);
        }
    }

    foreach (var kvp in demotionList)
    {
        Demote(kvp.Key, kvp.Value);
    }
}
```

```

private void AttemptDemotion(TKey key,
                            ValueContainer< TValue > container)
{
    long now = Stopwatch.GetTimestamp();
    var age = CalculateTimeSpan(container.additionTime, now);
    if (age > this.maxAgeBeforeDemotion)
    {
        Demote(key, container);
    }
}

private void Demote(TKey key,
                    ValueContainer< TValue > container)
{
    ValueContainer< TValue > oldContainer;
    this.strongReferences.TryRemove(key, out oldContainer);
    container.demoteTime = Stopwatch.GetTimestamp();
    var weakRef =
        new WeakReference< ValueContainer< TValue > >(container);
    this.weakReferences.AddOrUpdate(key,
                                    weakRef,
                                    (k, oldRef) => weakRef);
}

private static TimeSpan CalculateTimeSpan(long offsetA,
                                            long offsetB)
{
    long diff = offsetB - offsetA;
    double seconds = (double)diff / Stopwatch.Frequency;
    return TimeSpan.FromSeconds(seconds);
}
}

```

Example: Secondary Index

This example uses weak references to make updates to a simple database more efficient by avoiding immediate, potentially expensive index updates.

```
class Person
```

```
{  
    public string Id { get; set; }  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public DateTime Birthday { get; set; }  
}  
  
class PersonDatabase  
{  
    private Dictionary<string, Person> index =  
        new Dictionary<string, Person>();  
    private Dictionary<DateTime,  
        List<WeakReference<Person>>>  
        birthdayIndex =  
        new Dictionary<DateTime, List<WeakReference<Person>>>();  
  
    public bool NeedsIndexRebuild { get; private set; }  
  
    public void AddPerson(Person person)  
    {  
        this.index[person.Id] = person;  
        List<WeakReference<Person>> birthdayList;  
        if (!this.birthdayIndex.TryGetValue(person.Birthday,  
                                         out birthdayList))  
        {  
            birthdayIndex[person.Birthday]  
                = birthdayList  
                = new List<WeakReference<Person>>();  
        }  
  
        birthdayList.Add(new WeakReference<Person>(person));  
    }  
  
    public void RemovePerson(string id)  
    {  
        index.Remove(id);  
    }  
  
    public bool TryGetById(string id, out Person person)  
    {  
        return this.index.TryGetValue(id, out person);  
    }  
}
```

```

public bool TryGetByBirthday(DateTime birthday,
                           out List<Person> people)
{
    people = null;
    List<WeakReference<Person>> weakPeople;
    if (this.birthdayIndex.TryGetValue(birthday,
                                      out weakPeople))
    {
        var list = new List<Person>(weakPeople.Count);
        foreach(var wp in weakPeople)
        {
            Person person;
            if (wp.TryGetTarget(out person))
            {
                list.Add(person);
            }
            else
            {
                // we got a null reference --
                // we need to rebuild the indexes
                this.NeedsIndexRebuild = true;
            }
        }
        if (list.Count > 0)
        {
            people = list;
            return true;
        }
    }
    return false;
}
}

```

Object Resurrection

There is an overload of `WeakReference<T>`'s constructor that takes a Boolean value called `trackResurrection`:

```

WeakReference<MyObject> weakRef =
    new WeakReference<MyObject>(myObj, trackResurrection: true);

```

Resurrection is when you do something like this in a class's finalizer:

```
class MyObject
{
    static MyObject myObj;

    ~MyObject()
    {
        myObj = this;
    }
}
```

By doing this, you are taking an object that had no more references to it (hence why the finalizer ran) and re-adding a reference to it. This technique is sometimes used in advanced caching scenarios, but it has a number of drawbacks:

- The object has already been promoted to gen 1 by the garbage collector, and will never be demoted to an earlier generation.
- You must call `GC.ReRegisterForFinalizer` on an object or the finalizer will not run again for it.
- The state of the object can be indeterminate. Objects with native resources will have released them and they will need to be reinitialized. It can be tricky working through the exact object state.
- Any objects that the resurrected object refers to are also resurrected. If any of those objects have finalizers, they will also have already run, making your state trickier.

You should just consider this technique a bug unless you really understand the state of the objects you are dealing with. There are better ways to reuse objects.

If you do use it, then you can tell `WeakReference<T>` to allow longer access to the underlying object. If your object does not have a finalizer, then this parameter has no effect.

Dynamically Allocate on the Stack

Instead of allocating memory from the heap, it is possible to allocate dynamically sized buffers on the stack using `stackalloc`. Such allocations are faster than heap allocations and incur no garbage collection. However, there are some significant caveats:

- Doing so is explicitly unsafe. Instead of a managed array, you receive back a pointer to the beginning of the buffer and you are responsible for ensuring you do not exceed its bounds.
- You are severely limited in how much data you can allocate. Managed stacks are typically limited to 1 megabyte in size (or as little as 256KB in ASP.NET/IIS). Each stack frame takes some of that, and some frameworks can have very deep stacks.

To demonstrate how `stackalloc` works, see the `StackAlloc` sample program, which contains this code:

```
private static unsafe void DoStackAlloc(int size)
{
    int* buffer = stackalloc int[size];
    for (int i = 0; i < size; i++)
    {
        buffer[i] = i;
    }
}
```

The rest of the program runs this code in a loop, asking for input for how much to allocate. A sample run looks like this:

```
Enter size to stackalloc ('q' to exit): 100
Allocated 100-size array
Enter size to stackalloc ('q' to exit): 200
Allocated 200-size array
Enter size to stackalloc ('q' to exit): 100000
```

```
Allocated 100000-size array
Enter size to stackalloc ('q' to exit): 1000000
```

Process is terminated due to StackOverflowException.

StackOverflowException has the notable distinction of being an exception that your program cannot catch. The sample code wraps the allocation in an exception handler, but to no avail. When this exception is thrown, your application will immediately exit. If you run under a debugger, however, it can catch it.

Despite the risks and limitations, `stackalloc` is a valuable tool when you want small, dynamically sized arrays in your methods without the overhead of a heap allocation.

Investigating Memory and GC

In this section, you will learn many tips and techniques to investigate what is happening on the GC heap. In many cases, multiple tools can give you the same information. I will endeavor to describe the use of a few in each scenario, where applicable.

Performance Counters

.NET supplies a number of Windows performance counters, all in the .NET CLR Memory category. All of these counters except for Allocated Bytes/sec are updated at the end of a collection. If you notice values getting stuck, it is likely because collections are not happening very often.

- **# Bytes in all Heaps:** Sum of all heaps, except gen 0 (see the description for Gen 0 heap size).
- **# GC Handles:** Number of handles in use.

- **# Gen 0 Collections:** Cumulative number of gen 0 collections since process start. Note that this counter is also incremented for gen 1 and 2 collections because higher generation collections always imply collections of the lower generations as well.
- **# Gen 1 Collections:** Cumulative number of gen 1 collections since process start. Note that this counter is incremented for gen 2 collections as well because a gen 2 collection implies a 1 collection.
- **# Gen 2 Collections:** Cumulative number of gen 2 collections since process start.
- **# Induced GC:** Number of times `GC.Collect` was called to explicitly start garbage collection.
- **# of Pinned Objects:** Number of pinned objects the garbage collector observes during collection.
- **# of Sink Blocks in use:** Each object has a header that can store limited information, such as a hash code, or synchronization information. If there is any contention for use of this header, a sync block is created. These blocks are also used for interop metadata. A high counter value here can indicate lock contention. Yes, this counter's name is misspelled (look at the description in PerfMon).
- **# Total committed Bytes:** Number of bytes the garbage collector has allocated that are actually backed by the paging file.
- **# Total reserved Bytes:** Number of bytes reserved by garbage collector, but not yet committed.
- **% Time in GC:** Percentage of time the processor has spent in the GC threads compared to the rest of the process since the last collection. This counter does not account for background GC.
- **Allocated Bytes/sec:** Number of bytes allocated on a GC heap per second. This counter is not updated continuously, but only when a garbage collection starts.

- **Finalization Survivors:** Number of finalizable objects that survived a collection because they are waiting for finalization (which only happens in gen 1 collections). Also, see the Promoted Finalization-Memory from Gen 0 counter.
- **Gen 0 heap size:** Maximum number of bytes that can be allocated in gen 0, not the actual number of bytes allocated.
- **Gen 0 Promoted Bytes/Sec:** The rate of promotion from gen 0 to gen 1. You want this number to be as low as possible, indicating short memory lifetimes.
- **Gen 1 heap size:** Number of bytes in gen 1, as of the last garbage collection.
- **Gen 1 Promoted Bytes/Sec:** Rate of promotion from gen 1 to gen 2. A high number here indicates memory having a very long lifetime, and good candidates for pooling.
- **Gen 2 heap size:** Number of bytes in gen 2, as of the last garbage collection.
- **Large Object Heap Size:** Number of bytes on the large object heap.
- **Promoted Finalization Memory from Gen 0:** Total number of bytes that were promoted to gen 1 because an object somewhere in their tree is awaiting finalization. This is not just the memory from finalizable objects directly, but also the memory from any references those objects hold.
- **Promoted Memory from Gen 0:** Number of bytes promoted from gen 0 to gen 1 at the last collection.
- **Promoted Memory from Gen 1:** Number of bytes promoted from gen 1 to gen 2 at the last collection.

ETW Events

The CLR publishes numerous events about GC behavior. In most cases, you can rely on the tools to analyze these in aggregate for you, but it is still useful to understand how this information is logged in case you need to track down specific events and relate them to other events in your application. You can examine these in detail in PerfView with the Events view. Here are some of the most important:

- **GCStart_V1:** Garbage collection has started. Fields include:
 - Count: The number of collections that have occurred since the process began.
 - Depth: Which generation is being collected.
 - Reason: Why the collection was triggered.
 - Type: Blocking, background, or blocking during background.
- **GCEnd_V1:** Garbage collection has ended. Fields include:
 - Count, Depth: Same as for GCStart.
- **GCHeapStats_V1:** Shows stats at the end of a garbage collection.
 - There are many fields, describing all aspects of the heap such as generation sizes, promoted bytes, finalization, handles, and more.
- **GCCreateSegment_V1:** A new segment was created. Fields include:
 - Address: Address of the segment
 - Size: Size of the segment
 - Type: Small or large object heap
- **GCFreeSegment_V1:** A segment was released. Just one field:
 - Address: Address of the segment

- **GCAssignmentTick_V2:** Emitted every time about 100KB (cumulative) has been allocated. Fields include:
 - AllocationSize: Exact size of the allocation that triggered the event.
 - Kind: Small or large object heap allocation.
- **GCFinalizersBegin_V1:** Finalizers are starting to run.
- **GCFinalizersEnd_V1:** Finalizers are done running.
 - Count: Number of finalizers executed.
- **GCCreateConcurrentThread_V1:** A concurrent garbage collection thread was created.
- **GCTerminateConcurrentThread_V1:** A concurrent garbage collection thread was terminated.
- **GCSuspendEE_V1:** Threads are starting to suspend.
 - Reason: Why the suspension was initiated.
 - Count: GC Count at the time of this event.
- **GCSuspendEEEEnd_V1:** Threads are done suspending.
- **GCRestartEEBegin_V1:** Threads start resuming.
- **GCRestartEEEEnd_V1:** Threads are done resuming.

The order of events that are received is important. For a normal, foreground GC of any generation, the sequence is:

1. GCSuspendEE_V1: Start suspending threads.
2. GCSuspendEEEEnd_V1: All threads are suspended.
3. GCStart_V1: GC begins.

4. GCEnd_V1: GC work done.
5. GCRestartEEBegin_V1: Start resuming threads.
6. GCRestartEEEEnd_V1: All threads resumed. GC complete.

If you want to analyze these events in your own applications or utilities, see the sections on TraceEvent and PerfView in Chapter 8 for an easy-to-use library. Through judicious use and analysis of ETW events you can detect whether operations in your application are being affected by GC (or any other type of external influence).

What Does My Heap Look Like?

WinDbg can give you a few different views of the heap. First, by segment:

```
!eeheap -gc
```

The output will look something like this:

```
Number of GC Heaps: 1
generation 0 starts at 0x05824e2c
generation 1 starts at 0x0532100c
generation 2 starts at 0x05321000
ephemeral segment allocation context: none
    segment      begin      allocated      size
05320000  05321000  05891ff4  0x570ff4(5705716)
Large object heap starts at 0x06321000
    segment      begin      allocated      size
06320000  06321000  07312c80  0xff1c80(16718976)
07900000  07901000  088ee660  0xfed660(16701024)
08a30000  08a31000  09a1e660  0xfed660(16701024)
09c80000  09c81000  0ac6e660  0xfed660(16701024)
0ac80000  0ac81000  0bc6e540  0xfed540(16700736)
```

...more segments...

Total Size: Size: 0x213b9d94 (557555092) bytes.

GC Heap Size: Size: 0x213b9d94 (557555092) bytes.

If the process is running server GC, there will be more than one heap, each with their own set of ephemeral, gen 2, and large object segments.

Another view is provided with the !HeapStat command, which aggregates across all segments to break down the sizes of each generation, including free space.

0:007> !HeapStat

Heap	Gen0	Gen1	Gen2	LOH
Heap0	446920	5258784	12	551849376

Free space:

				Percentage
Heap0	12	1948	0	15936SOH: 0% LOH: 0%

This output shows that there is very little in the gen 2 heap, and there is an insignificant amount of free space (fragmentation) on the heap. The letters SOH mean “Small Object Heap”, which means every segment other than the Large Object Heap segments.

The !VMMap command shows information about virtual address regions and the levels of protection applied to them:

0:000> !VMMap

Start	Stop	Length	AllocProtect	Protect	State	Type
00000000-00f5ffff	00f60000			NA	Free	
00f60000-00f60fff	00001000	ExWrCp	Rd	Commit	Image	
00f61000-00f61fff	00001000	ExWrCp		Reserve	Image	
00f62000-00f62fff	00001000	ExWrCp	Rd	Commit	Image	
00f63000-00f63fff	00001000	ExWrCp		Reserve	Image	
00f64000-00f64fff	00001000	ExWrCp	Rd	Commit	Image	
00f65000-00f65fff	00001000	ExWrCp		Reserve	Image	

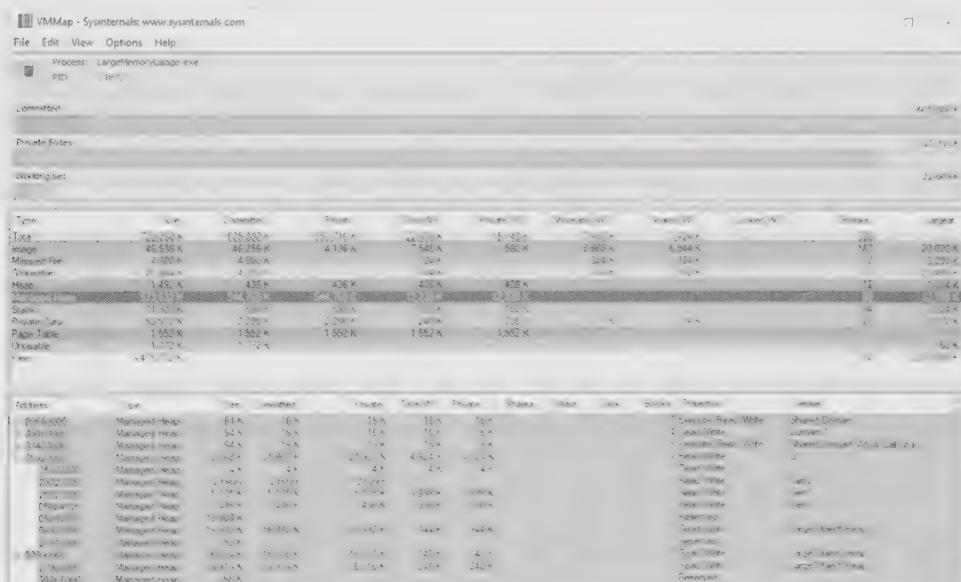
00f66000-00f66fff	00001000	ExWrCp	Rd	Commit	Image
00f67000-00f67fff	00001000	ExWrCp		Reserve	Image
...					

The !VMStat command will take that information and summarize it by State:

0:000> !VMStat					
TYPE	MINIMUM	MAXIMUM	AVERAGE	BLK COUNT	TOTAL
Free:					
Small	8K	64K	43K	30	1,315K
Medium	84K	996K	332K	10	3,323K
Large	1,152K	2,090,816K	204,209K	17	3,471,563K
Summary	8K	2,090,816K	60,986K	57	3,476,203K
Reserve:					
Small	4K	64K	34K	34	1,183K
Medium	68K	1,012K	299K	56	16,779K
Large	1,376K	32,768K	12,073K	7	84,515K
Summary	4K	32,768K	1,056K	97	102,479K
Commit:					
Small	4K	64K	12K	204	2,575K
Medium	68K	964K	347K	44	15,307K
Large	1,048K	16,332K	12,716K	47	597,671K
Summary	4K	16,332K	2,086K	295	615,555K
Private:					
Small	4K	64K	19K	88	1,716K
Medium	68K	1,012K	285K	57	16,267K
Large	1,376K	32,768K	15,215K	41	623,851K
Summary	4K	32,768K	3,450K	186	641,835K
Mapped:					
Small	4K	64K	25K	8	204K

Medium	68K	1,004K	374K	6	2,247K
Large	1,540K	18,320K	5,442K	5	27,211K
Summary	4K	18,320K	1,561K	19	29,663K
Image:					
Small	4K	64K	12K	142	1,839K
Medium	68K	964K	366K	37	13,571K
Large	1,048K	15,712K	3,890K	8	31,124K
Summary	4K	15,712K	248K	187	46,535K

The SysInternals tool VMMap can also give you a good summary of all the segments in a process. Once you have selected the process, highlight the Managed Heap in the table, and you will see a list of all segments in the process.



How Long Does a Collection Take?

The GC records many events about its operation. You can use PerfView to examine these events in a very efficient way.

To see statistics on GC, start the AllocateAndRelease sample program.

Start PerfView and follow these steps:

1. Menu Collect | Collect (Alt+C).
2. Expand Advanced Options. You can optionally turn off all event categories except GC Only, but for now just leave the default selection, as GC events are included in .NET events.
3. Check No V3.X NGEN Symbols. (This will make symbol resolution faster.)
4. Click Start.
5. Wait for a few minutes while it measures the process's activity. See Chapter 1 for a discussion of how many samples you need to collect. (If collecting for more than a few minutes, you may want to turn off CPU events.)
6. Click Stop Collection.
7. Wait for the files to finish merging.
8. In the resulting view tree, double-click on the GCStats node, which will open up a new view.
9. Find the section for your process.

For each process, you will find a list of data points and a set of tables summarizing GC behavior.

At the top of each section is a list of items describing the overall information.

Below that, you will find a table summarizing all the generations of GC.

GC Trace Summary Item	Description
CommandLine	The exact command that executed the process
Runtime Version	The version of the CLR that is executing
CLR Startup Flags	Flags controlling the behavior of the GC, such as CONCURRENT_GC or SERVER GC
Total CPU Time	Total time, in milliseconds, taken by the process during the profile
Total GC CPU Time	Total time, in milliseconds, spent doing garbage collection
Total Allocs	Amount of allocation you have done
GC CPU MSec/MB Alloc	How much time, in milliseconds, the GC spent processing 1 MB of memory
Total GC Pause	Amount of time, in milliseconds, the process was paused for GC
% Time paused for Garbage Collection	GC pause time, expressed as a percent of total CPU time
% CPU Time spent Garbage Collecting	CPU time can be different than % Time if you are running server GC
Max GC Heap Size	Maximum size of the GC heap during profiling
Peak Process Working Set	Maximum size of the working set during profiling
Peak Virtual Memory Usage	Maximum virtual memory reserved during profiling

GC Summary Info Column	Description
Gen	Generation, including ALL, which aggregates all GCs into a single set of stats.
Count	Number of collections.
Max Pause	Longest time, in milliseconds, that GC was paused.
Max Peak MB	Maximum size of the generation on the heap.
Max Alloc MB/sec	Peak allocation rate.
Total Pause	Sum of all pause times, in milliseconds.
Total Alloc MB	Amount of memory allocated.
Alloc MB/MSec GC	Amount of memory allocated per millisecond of GC time. This is a measure of GC efficiency. Higher numbers mean a more effective (or less intrusive) GC.
Survived MB/MSec GC	Amount of memory that survives a GC, per millisecond of GC time. This is another measure of GC efficiency. Higher numbers mean more memory is surviving.
Mean Pause	Average pause time, in milliseconds.
Induced	Count of explicit GC invocations (GC.Collect).

GC Details Column	Description
GC Index	The order in which the GC occurred
Pause Start	Time stamp, in milliseconds, from when the profile started, of when the GC occurred
Trigger Reason	Reason the GC happened.
Gen	Generation and letter code indicating the type of GC. Gen is 0-2. N=NonConcurrent, B=Background, F=Foreground, I=Induced, i=induced, not forced.
Suspend Msec	The number of milliseconds required to suspend running threads
Pause Msec	Total time process is paused for GC
% Pause Time	% of time in GC, since previous GC
% GC	% of CPU time used by GC
Gen0 Alloc MB	Amount allocated since previous GC
Gen0 Alloc Rate MB/sec	Allocation rate since previous GC
Peak MB	Peak size of heap during GC
After MB	Size of heap after GC is complete
Ratio Peak/After	Efficiency, higher is better
Promoted MB	Amount of memory that survived the GC
Gen0 MB	Gen 0 size after this GC is complete
Gen0 Survival Rate %	% of objects in gen 0 that survived GC
Gen 0 Frag %	% of gen 0 that is free space
Gen 1 MB	Gen 1 size after this GC is complete
Gen1 Survival Rate %	% of objects in gen 1 that survived GC
Gen1 Frag %	% of generation 1 that is free space
Gen2 MB	Gen 2 size after this GC is complete
Gen2 Survival Rate %	% of objects in gen 2 that survived GC
Gen2 Frag %	% of gen 2 that is free space
LOH MB	LOH size after this GC is complete
LOH Survival Rate %	% of objects on LOH that survived GC
LOH Frag %	% of the LOH that is free space
Finalizable Surv MB	Finalizable object size that survived GC
Pinned Obj	# of pinned objects this GC promoted. Fewer is better.

GC Rollup By Generation										
All times are in msec.										
Gen Count	Max Pause	Max Peak MB	Max Alloc MB/sec	Total Pause	Total Alloc MB	Alloc MB/MSec GC	Survived MSec GC	MB/Mean Pause	Induced	
ALL	12651	12.3	5.5	291.128	1,495.3	25,648.2	17.2	Infinity	0.1	0
0	11780	0.7	5.4	261.359	716.6	23,816.5	0.0	Infinity	0.1	0
1	0	0.0	0.0	0.000	0.0	0.0	0.0	Nan	Nan	0
2	871	12.3	5.5	291.128	778.7	1,831.7	0.0	Infinity	0.9	0

Figure 2.9. The GCStats table for the AllocateAndRelease sample program. This shows you the number of GCs that occurred as well as interesting stats like the mean/max pause times, and allocation rates.

Below this table, you will find even more detailed tables listing specific GC instances in various categories, such as “Pauses > 200 MSec”, “LOH Allocation Pause (due to background GC) > 200 MSec”, “Gen 2”, and “All GC Events”.

As you can see, there is a wealth of information with each GC event which you can use to analyze GC performance.

Where Are My Allocations Occurring?

Visual Studio can track .NET memory allocations via ETW sampling. Note that this is completely different than the Memory Usage profiler. That report is essentially a heap dump analyzer, which shows static snapshots of the objects on the heap and their ownership references, back to the root. The .NET memory allocation report in Performance Wizard uses ETW events to track which methods are actually doing the allocation, regardless of who ends up holding onto the references. It uses the GCAccumulationTick_V2 ETW event that the CLR emits every 100KB of allocations.

Clicking on a method name will again take you to the familiar Function Details view. Just remember that you are looking at memory allocations rather than CPU time.

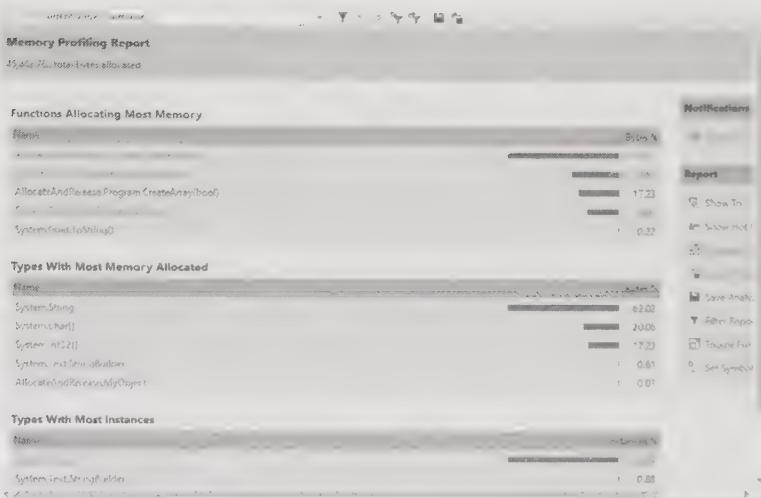


Figure 2.10. Memory Profiling Report, which shows which methods allocate the most, as well as which types take up the most memory.

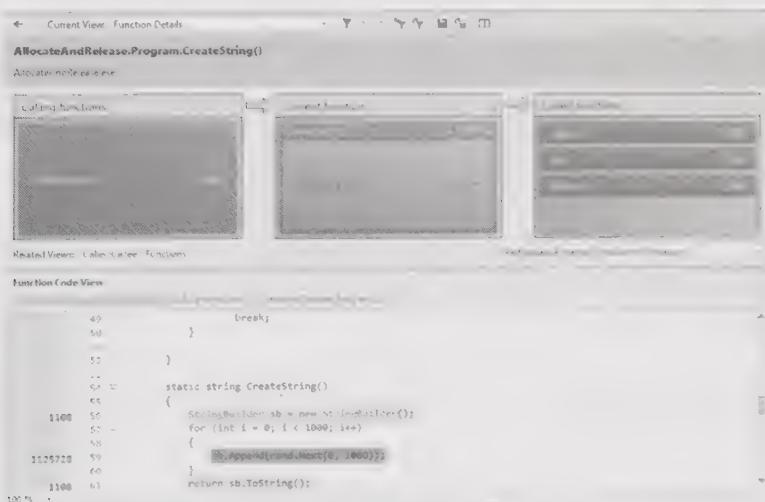


Figure 2.11. Memory allocation function details, showing which called methods, as well as source lines, are responsible for allocations.

This report has many other views to drill-down along different dimensions. The Allocation view in particular is interesting. It is what is shown when you click on a type name in the main summary view.

The screenshot shows a table of memory allocations. The columns are: Name, Inclusive Allocations, Exclusive Allocations, Inclusive Bytes, and Exclusive Bytes. The rows show allocations for System.String, System.Char[], System.IntPtr[], System.Text.StringBuilder, and AllocateAndRelease.MyObject. The total for System.String is 1,110,265 inclusive allocations and 28,433,778 bytes.

Name	Inclusive Allocations	Exclusive Allocations	Inclusive Bytes	Exclusive Bytes	Inclusive Allocati
System.String	1,110,265	1,110,265	28,433,778	28,433,778	
AllocateAndRelease.Program.Main(string[])	1,110,265	0	28,433,778	0	
AllocateAndRelease.Program.DoAnAllocation()	1,110,265	0	28,433,778	0	
AllocateAndRelease.Program.CreateString()	1,109,108	1,100,000	28,334,276	21,915,382	
AllocateAndRelease.MyObject..ctor()	1,157	0	99,502	0	
System.Char[]	9,972	9,972	9,196,400	9,196,400	
System.IntPtr[]	1,081	1,081	7,900,972	7,900,972	
System.Text.StringBuilder	9,972	9,972	279,216	279,216	
AllocateAndRelease.MyObject	1,157	1,157	32,396	32,396	

Figure 2.12. A summary of allocations by type, rather than method stack.

This view aggregates by type and shows you which methods contribute to their allocations most frequently.

Another option is PerfView, which can show you the same information as Visual Studio, and much more, though the interface is not quite as polished.

1. With PerfView, collect either .NET or just GC Only events.
2. Once completed, open the GC Heap Alloc Stacks view and select the desired process from the process list. (For a simple example, use the AllocateAndRelease sample program) from the process list.)
3. On the By Name tab, you will see types sorted in order of total allocation size. Double-clicking a type name will take you to the Callers tab, which shows you the stacks that made the allocations.

See Chapter 1 for more information on using PerfView's interface to get the most out of the view.

Using the above information, you should be able to find the stacks for all the allocations that occur in the test program, and their relative frequency. For example, in my trace, string allocation accounts for roughly 59.5% of all memory allocations.

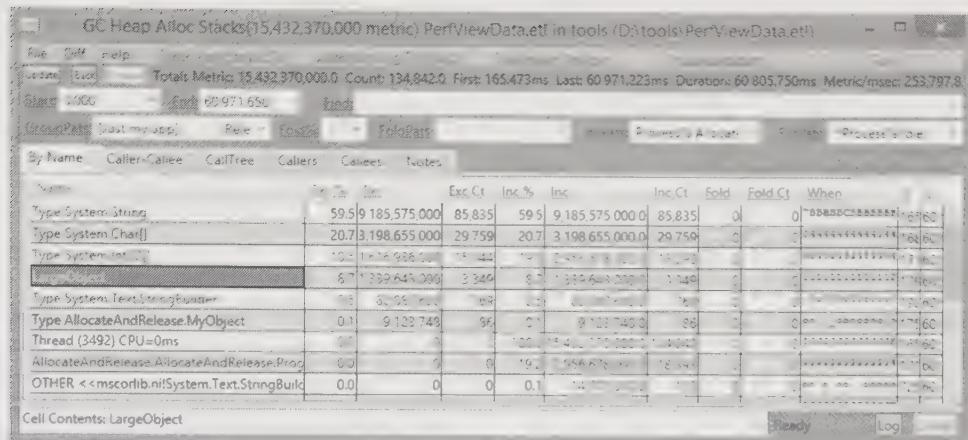


Figure 2.13. The GC Heap Alloc Stacks view shows the most common allocations in your process. The LargeObject entry is a pseudo node; double-clicking on it will reveal the actual objects allocated on the LOH.

You can also use CLR Profiler to find this information and display it in a number of ways.

Once you have collected a trace and the Summary window opens, click on the Allocation Graph button to open up a graphical trace of object allocations and the methods responsible for them.

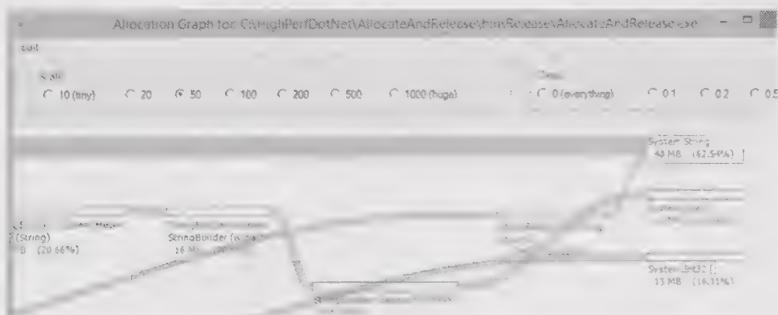


Figure 2.14. CLR Profiler's visual depiction of object allocation stacks quickly points you to objects you need to be most concerned about.

The most frequently allocated objects are also the ones most likely to be triggering garbage collections. Reduce these allocations and the rate of GCs will go down.

What Are All The Objects On The Heap?

All of the tools in this section will use the LargeMemoryUsage sample program, reproduced here:

```
class Program
{
    const int ArraySize = 1000;
    static object[] staticArray = new object[ArraySize];

    static void Main(string[] args)
    {
        var localArray = new object[ArraySize];

        var rand = new Random();
        for (int i = 0; i < ArraySize; i++)
        {
            staticArray[i] = GetNewObject(rand.Next(0, 4));
            localArray[i] = GetNewObject(rand.Next(0, 4));
        }

        Console.WriteLine("Examine heap now. Press any key to exit.");
        Console.ReadKey();

        // This will prevent localArray from being
        // garbage collected before you take the snapshot
        Console.WriteLine(staticArray.Length);
        Console.WriteLine(localArray.Length);
    }

    private static Base GetNewObject(int type)
    {
        Base obj = null;
        switch (type)
        {
            case 0: obj = new A(); break;
```

```

        case 1: obj = new B(); break;
        case 2: obj = new C(); break;
        case 3: obj = new D(); break;
    }
    return obj;
}

class Base
{
    private byte[] memory;
    protected Base(int size) { this.memory = new byte[size]; }
}

class A : Base { public A() : base(1000) {} }
class B : Base { public B() : base(10000) {} }
class C : Base { public C() : base(100000) {} }
class D : Base { public D() : base(1000000) {} }

```

This simple program just allocated random amounts of different classes and waits for you to analyze the heap before exiting.

There are a number of ways to analyze this heap, starting with a very low level.

Using WinDbg, you could execute the !DumpHeap command to just dump a list of every single object on the heap:

```

0:007> !DumpHeap
Address      MT      Size
02aa1000 00b2ac70      10 Free
02aa100c 00b2ac70      10 Free
02aa1018 00b2ac70      10 Free
02aa1024 71911eac      84
02aa1078 71912000      84
02aa10cc 71912044      84
02aa1120 71912088      84
02aa1174 719120cc      84
02aa11c8 719120cc      84
02aa121c 71912104      12
02aa1228 71911d64      14

```

The MT column specifies the address of the Method Table, which is essentially equivalent to the class.

You can dump a specific object to get its information:

```
0:007> !DumpObj /d 02bb8cf4
Name:          LargeMemoryUsage.A
MethodTable:   00de4f6c
EEClass:       00de196c
Size:          12(0xc) bytes
File:          D:\SampleCode\...\LargeMemoryUsage.exe
Fields:
    MT      Field Offset          Type VT     Attr     Value Name
719160e8 4000003      4 System.Byte[] 0 instance 02bb8d00 memory
```

Dumping every object in the heap will usually be overwhelming. Thankfully, you can filter the output a bit, such as by type:

```
0:007> !DumpHeap -type LargeMemoryUsage.A
Address      MT      Size
02aab98 00de4f6c      12
02ab82cc 00de4f6c      12
02ab86cc 00de4f6c      12
02ab8acc 00de4f6c      12
```

If you have a heap range, you can filter the output to only objects within a specified range:

```
!DumpHeap -type LargeMemoryUsage.A 02aab98 02ab86cc
```

While there is a rudimentary scripting language in WinDbg, doing advanced heap analysis can be difficult. Another option is to use CLR MD to analyze the objects.

DumpHeap Parameter	Description
-min	Display objects at least the given size.
-max	Display objects at most the given size.
-startatLowerBound	Start scanning the heap at the specified address (must be the address of an object).
-type	Does a substring match of the argument against the type name.
-mt	Displays only objects with the given method table address. This is a more precise way to get output of a specific type of objects compared to -type, which can match on different types.
-short	Outputs object addresses only.
-strings	Displays a summary of strings in the heap.
-stat	Only displays the statistical summary.

```
private static void PrintGen0Objects(ClrRuntime clr)
{
    var heap = clr.Heap;

    foreach(var obj in heap.EnumerateObjects())
    {
        Console.WriteLine($"0x{obj.Address:x} - {obj.Type.Name}");
    }
}
```

Because you have programmatic access to the same properties of an object as in WinDbg, you can filter by the same criteria, or even come up with more complex criteria to find and analyze objects.

So far, we have looked at ways to analyze each discrete object. That certainly comes in handy while debugging, but often when we are analyzing overall behavior, we want to consider all of the objects in aggregate.

Starting with Visual Studio 2013 Premium Edition (Enterprise Edition starting in Visual Studio 2015), there is a managed heap analyzer. You can access it after opening a managed memory dump by selecting “Debug Managed Memory”.

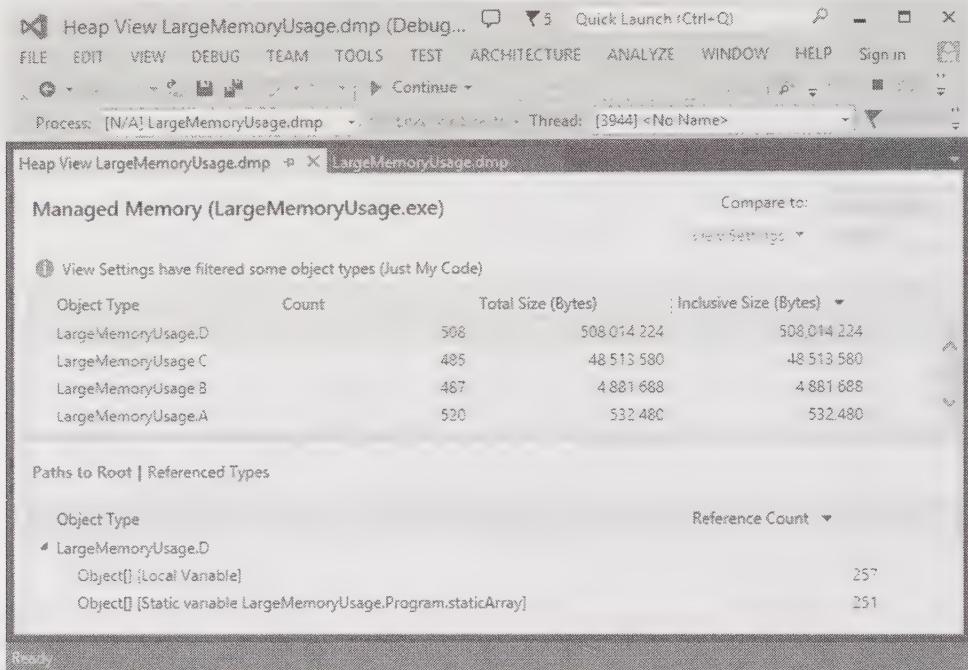


Figure 2.15. Higher editions of Visual Studio include this heap analysis view, which works from managed memory dumps.

From here, you can do three things:

1. See all instances of that type (double-click on the type name)
2. See the various roots of the objects (what is keeping them in memory)
3. See what other types are referenced by the highlighted type

There is also a feature in the Performance Profiler to get heap snapshots during runtime. To access this, go to the Analyze | Performance Profiler, then select Memory Usage. The output is a graph of memory usage against garbage collections. While the analysis is running, you can take snapshots of the heap whenever you want.

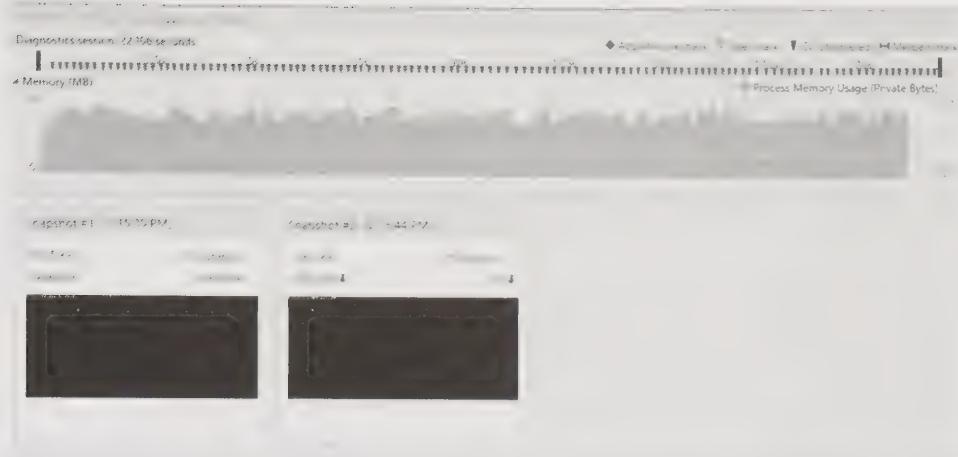


Figure 2.16. Memory usage graph, showing aggregate memory usage, GCs, and heap snapshots.

Clicking the size or object count in a snapshot will take you to a table of all the objects on the heap, and their paths to the root (what is keeping them alive).

Each snapshot also allows you to see just the objects that changed from the previous snapshot, helping you analyze allocations over time.

These options give you a fairly basic, but useful overview of your heap. If you need more analytical power, then I recommend PerfView. PerfView will not show you individual objects, but its ability to show aggregated object relationships is unparalleled.

To use this feature in PerfView:

1. From the Memory menu, select Take Heap Snapshot. Note that this does not pause the process (unless you check the option to Freeze), but it does have a significant impact on the process's performance.
 2. Highlight the desired process in the resulting dialog.
 3. Click Dump GC Heap.
 4. Wait until collection is finished, then close the window.

CHAPTER 2. MEMORY MANAGEMENT

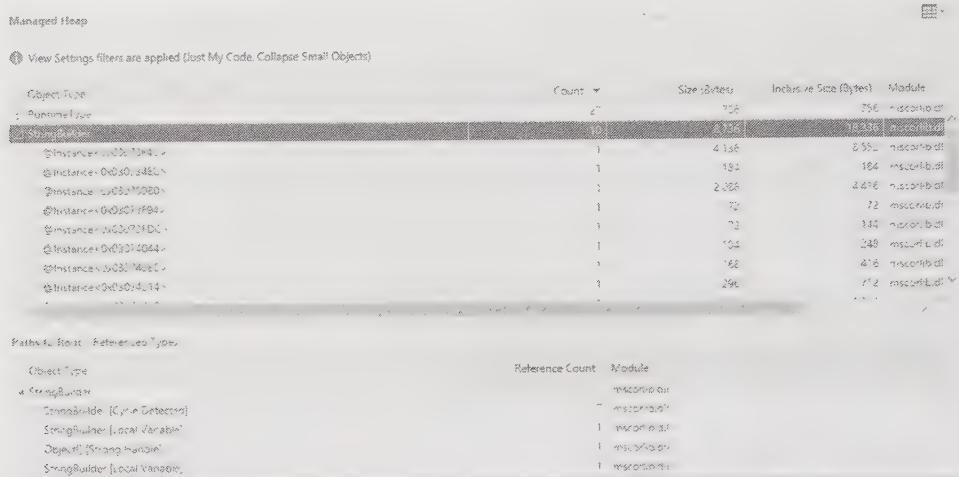


Figure 2.17. Visual Studio’s heap snapshot view. This shows the various paths to root of each object type in aggregate, or with specific instances.

5. Open the file from the PerfView file tree (it may automatically open for you when you close the collection window).

You should see a table like this:

Name	Exc %	Exc ?	Exc Ct ?	Inc %	Inc ?
[LargeMemoryUsage!LargeMemoryUsage.D]	88.5	462,013,000	924	88.5	462,013,000.0
LargeMemoryUsage!LargeMemoryUsage.C	10.4	54,213,010	1,084	10.4	54,213,010.0
LargeMemoryUsage!LargeMemoryUsage.B	1.0	5,242,552	1,046	1.0	5,242,552.0
LargeMemoryUsage!LargeMemoryUsage.A	0.1	484,352	946	0.1	484,352.0
[Pinned handle]	0.0	21,586	153	0.0	22,926.0
[local var]	0.0	4,032	3	49.5	258,117,400.0
[static var LargeMemoryUsage.Program.staticArray]	0.0	4,016	2	50.5	263,847,600.0

Figure 2.18. A PerfView trace of the largest objects in the heap.

It tells you immediately that D accounts for 88% of the program’s memory at 462 MB with 924 objects. You can also see local variables are holding on to 258 MB of memory and the `staticArray` object is holding onto 263 MB of memory.

PerfView is somewhat unique in that you can control how the sub-objects contribute to the size of their parent objects. This is done with the folding configuration. You can specify a folding percentage, below which all memory is at-

tributed to the parent object, or a folding pattern to specify that certain object types are always folded into their parent objects (they effectively disappear from analysis). See Chapter 1 for more details on how to use PerfView.

You can also get a graphical view of the same information with CLR Profiler. While the program is running, click the Show Heap Now button to capture a heap sample.

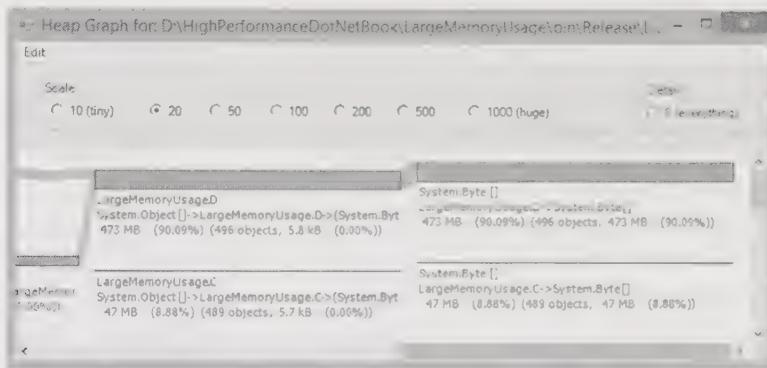


Figure 2.19. CLR Profiler shows you some of the same information as PerfView, but in a graphical format.

Where Is the Memory Leak?

There are many ways memory can leak, and all of the sections under “Investigating Memory and GC” in this chapter can help you narrow problems down, but there are a few general ways memory can leak in managed applications:

- Unexpected references are being held for objects, prohibiting the garbage collector from cleaning them up.
- Full garbage collections are rare, keeping a lot of memory in the process, but otherwise unused and unrooted. This is not usually a problem, and somewhat by design.
- There is high fragmentation, especially of the gen 2 or large object heap.

- A high number of pinned objects are preventing efficient collections. See later in this chapter for diagnosing extreme pinning situations.

In Visual Studio (Premium or Enterprise editions), you can open up the heap dump and debug the managed heap. When you click on a type, the tabs below will allow you to see which other types are referencing those objects.

You can use PerfView for more detailed analysis:

1. In the Memory menu, select Take Heap Snapshot.
2. In the resulting dialog box, select the process to analyze and click Dump GC Heap. You can optionally freeze the process or force a GC to occur before collecting the snapshot.
3. Once the snapshot is done, click Close.

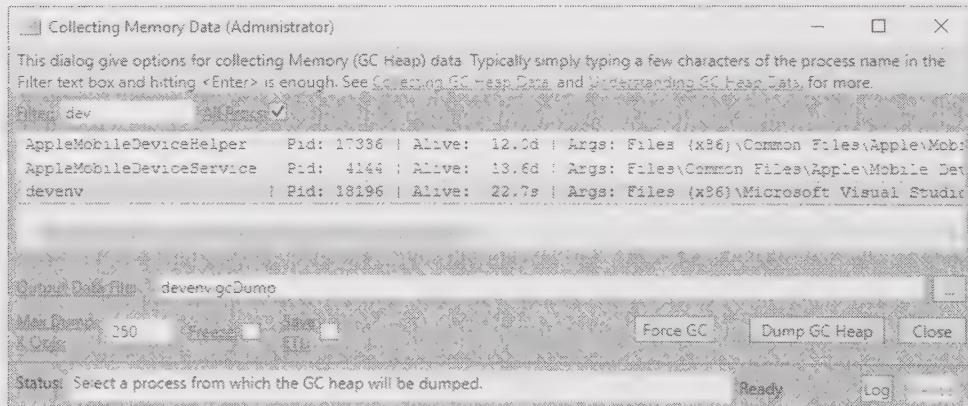


Figure 2.20. PerfView's Heap Snapshot dialog samples the managed heap for easy analysis.

Once the snapshot is completed, a file will show up in the left-hand pane. Double-click this file to open a view of the types in the heap. You can manipulate this view as any of the other stack views in PerfView (e.g., with grouping, folding, and filtering). You can double-click the entry for a type, which switches to the Referred-From view.

Objects that refer to LargeMemoryUsage!LargeMemoryUsage.D

Name ?	Inc % ?	Inc ?	Inc Ct ?	Exc % ?	Exc ?
✓ LargeMemoryUsage!LargeMemoryUsage.D	88.5	462,013,000.0	924	88.5	462,013,000
+ [static var LargeMemoryUsage.Program.staticArray]	44.8	234,006,300.0	468	0.0	0
+ [✓ [local var]]	43.7	229,006,800.0	456	0.0	0
+ ✓ [.NET Roots]	43.7	229,006,800.0	456	0.0	0
+ ✓ ROOT	43.7	228,006,800.0	456	0.0	0

Figure 2.21. PerfView shows the ownership of the objects on the heap, allowing easy leak analysis.

This view clearly shows that the D objects belong to the staticArray variable and a local variable (those lose their names during compilation).

You can generally get a good sense for what is on the heap from this view. If you take two dumps separated in time, then you can use the Diff menu to calculate a difference between the two snapshots. This can give you an idea for what is accumulating uncollected, if anything.

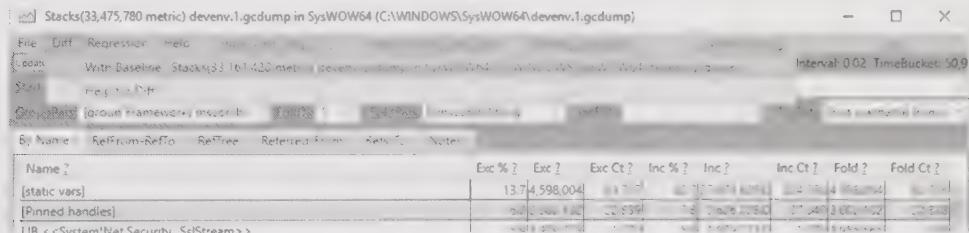


Figure 2.22. If you open two heap snapshots, you can compare them to get a view that shows differences.

Visual Studio and PerfView are mostly useful for aggregate analysis. PerfView is a sampling profiler, even when it analyzes the heap, so it will sometimes give a skewed picture of what the heap looks like. If you need to drill down onto a specific object, or get the absolute truth about the whole picture, then you need to start using the debugger or CLR MD.

In WinDbg, to get a quick summary of what is on the heap, run the !DumpHeap -stat command:

```
0:023> !DumpHeap -stat
```

CHAPTER 2. MEMORY MANAGEMENT

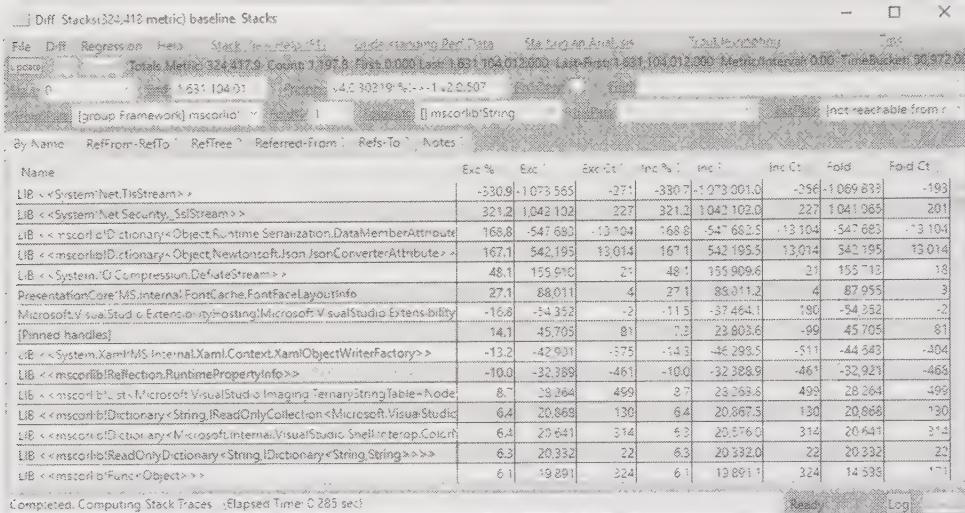


Figure 2.23. The Heap Diff view is the same as all stack views in PerfView, but the numbers will represent the difference of the target snapshot with the baseline.

```
...
71f718f8      8752      525120 System.Reflection.RuntimeMethodInfo
139e5424     15138      544968 System.Collections.Immutable.Sort...
71f7ffe4     11294      573796 System.Object[]
1370f7d0      4605      626280 Microsoft.VisualStudio.Compositio...
13707114     6190      990400 Microsoft.VisualStudio.Compositio...
1370f24c      5482     1227968 Microsoft.VisualStudio.Compositio...
71f8419c     4799     4684529 System.Byte[]
71f7fbf0    108732     8303452 System.String
00586810    30707     72014878     Free
```

It will produce a lot of output. I usually scroll to the end of the object summary to look at the largest consumers of heap space. (Note that after the object summary, it prints a list of objects that appear after free blocks – you want to scroll above that.)

If you do this a couple of times between letting the application run (and presumably leak), you can get a sense for what objects are taking up space. If you see the Free size increasing, that is an indication of either no collections happening or heap fragmentation. See later in this chapter for how to diagnose fragmentation.

The downside of WinDbg is that it is harder to get an overall picture of object ownership, especially for common objects like `System.Byte[]` or `System.String`. For this, use PerfView as described above.

If you want to analyze a single object, you will need to get its address first. To get the addresses of objects, use the `!DumpStackObjects` command, or use `!DumpHeap` to find objects of interest on heap, as in this example:

```
0:004> !DumpHeap -type LargeMemoryUsage.C
Address      MT      Size
021b17f0  007d3954        12
021b664c  007d3954        12
...
Statistics:
      MT   Count   TotalSize Class Name
007d3954     475       5700 LargeMemoryUsage.C
Total 475 objects
```

Once you have the object's address you can use the `!gcroot` command:

```
0:003> !gcroot 02ed1fc0
HandleTable:
 012113ec (pinned handle)
-> 03ed33a8 System.Object[]
-> 02ed1fc0 System.Random

Found 1 unique roots (run '!GCRoot -all' to see all roots).
```

`!gcroot` is often adequate, but it may miss some cases, in particular if your object is rooted from an older generation. For this, you will need to use the `!findroots` command.

In order for this command to work you first need to set a breakpoint in the GC, right before a collection is about to happen, which you can do by executing:

```
!findroots -gen 0  
g
```

This sets a breakpoint right before the next gen 0 GC happens. It then loses effect and you will need to run the command again to break on the following GC.

Once the code breaks, you need to find the object you are interested in and execute this command with its address:

```
!findroots 027624fc
```

If the object is already in a higher generation than the current collection generation, you will see output like this:

```
Object 027624fc will survive this collection:  
gen(0x27624fc) = 1 > 0 = condemned generation.
```

If the object itself is in the current generation being collected, but it has roots from an older generation, you will see something like this:

```
older generations::Root: 027624fc (object)->  
023124d4(System.Collections.Generic.List`1  
[[System.Object, mscorelib]])
```

If that is too tedious, you can use build your own `!gcroot` command using CLR MD.

```
const string TargetType = "LargeMemoryUsage.D";

private static void PrintRootsOfObjects(ClrRuntime clr)
{
    PrintHeader("Roots of Object");

    Dictionary<ulong, CClrObject> childToParents =
        new Dictionary<ulong, CClrObject>();
    var heap = clr.Heap;

    // Find an arbitrary object for demo purposes
    CClrObject targetObject = FindObjectOfType(clr, TargetType);

    if (targetObject.Address == 0)
    {
        Console.WriteLine(
            $"Could not find any objects of type {TargetType}");
        return;
    }

    // Analyze all objects, build up reference map
    foreach (var obj in heap.EnumerateObjects())
    {
        foreach (var objRef in obj.EnumerateObjectReferences())
        {
            childToParents[objRef.Address] = obj;
        }
    }

    // Walk up the chain of references
    CClrObject currentObj = targetObject;
    int indentSize = 0;
    while(true)
    {
        Console.Write(new string(' ', indentSize));
        Console.WriteLine(
            $"0x{currentObj.Address:x} - {currentObj.Type.Name}");

        CClrObject parentObject;
        if (!childToParents.TryGetValue(currentObj.Address,
                                       out parentObject))
        {
```

```
        break;
    }
    currentObj = parentObject;
    indentSize += 4;
}
}

private static CClrObject FindObjectOfType(CClrRuntime clr,
                                            string typeName)
{
    foreach (var obj in clr.Heap.EnumerateObjects())
    {
        if (obj.Type.Name == TargetType)
        {
            return obj;
        }
    }
    return new CClrObject();
}
```

This produces output similar to the following:

```
Roots of Object
=====
0x2e46bfc - LargeMemoryUsage.D
0x2e43428 - System.Object[]
```

How Big Are My Objects?

Calculating an object's size is a bit tricky. Do you mean the size of all the fields in that object? What if there is a reference to another object, such as an array is that included? What if two objects both refer to each other?

Thankfully, most tools that show object size follow an algorithm that has these concepts:

1. Exclusive size is the size of the object and all its fields. Referred objects are not included (but the size of the references to those objects, 4- or 8-bytes, are).
2. Inclusive size is the size of the object and all of the objects referred to by that object.
3. Object references are traced until they run out of references, or they touch an already-examined object. This avoids double-counting.

To get object sizes in Visual Studio, use the Memory Usage profiler:

1. Go to the Analyze | Performance profiler... menu option (or Alt + F2).
2. Select Memory Usage.
3. Execute the target program.
4. When desired, take a heap snapshot.
5. End profiling, or the target program.

Managed Heap					Filter
Object Type	Count	Size (Bytes)	Inclusive Size (Bytes)	Module	
▲ LargeMemoryUsage.B	502	6,024	5,032,048	LargeMemoryUsage.exe	
@Instance<0x051F44E4>	1	12	10,024	LargeMemoryUsage.exe	
@Instance<0x051F9770>	1	12	10,024	LargeMemoryUsage.exe	
@Instance<0x051FFE2C>	1	12	10,024	LargeMemoryUsage.exe	
Paths to Root : Referenced Objects					
Instance		Size (Bytes)	Inclusive Size (Bytes)	Module	
▲ LargeMemoryUsage.B @Instance<0x051F44E4>		12	10,024	LargeMemoryUsage.exe	
Byte[] (Bytes > 10K) @Instance<0x051F44E0>		10,012	10,012	mscorlib.dll	

Figure 2.24. Visual Studio's Memory Usage profiler can show aggregate and individual object sizes, including referenced objects.

If you do not see the level of detail you expect, make sure that the table's view options has turned off "Collapse Small Objects" and "Just My Code."

In WinDbg, there are a couple SOS command that can show the same information.

The !DumpObj command can show exclusive size of an object:

```
0:007> !DumpObj /d 058e8230
Name:           LargeMemoryUsage.D
MethodTable:    035d4e74
EEClass:        035d1870
Size:          12(0xc) bytes
File:          D:\HighPerformanceDotNetBook\...\LargeMemoryUsage.exe
Fields:
    MT      Field Offset          Type VT     Attr   Value Name
71b54080 4000003       4  System.Byte[]  0 instance 2a895510 memory
```

You can see that it does not take into account the owned byte array. For that, use the !ObjSize command:

```
0:007> !ObjSize 058e8230
sizeof(058e8230) = 1000028 (0xf425c) bytes (LargeMemoryUsage.D)
```

If you run !ObjSize without any parameters, it will show a list of all threads and GC handles, totaling up the size of objects rooted by each one.

```
0:007> !ObjSize
...
Thread 5580 (LargeMemoryUsage.Program.Main(System.String[])
[D:\HighPerformanceDotNetBook\...\Program.cs @ 29]):
ebp+1c: 012ff37c -> <exec cmd="!DumpObj /d 05383448">
05383448</exec>: 283846000 (0x10eb2570) bytes (System.Object[])
...
Handle (pinned): 035b13ec -> 06383510: 286744176 (0x11175e70) bytes
(System.Object[])
Handle (pinned): 035b13f0 -> 06382500: 8864 (0x22a0) bytes
```

```
(System.Object[])
Handle (pinned): 035b13f4 -> 063822e0: 640 (0x280) bytes
(System.Object[])
Handle (pinned): 035b13f8 -> 0538121c: 12 (0xc) bytes
(System.Object)
Handle (pinned): 035b13fc -> 06381020: 8440 (0x20f8) bytes
(System.Object[])
```

CLR MD can also calculate this size for you. You have to do the work of traversing the objects yourself.

```
private static void PrintObjectSize(ClrRuntime clr)
{
    PrintHeader("Object Size");

    var obj = FindObjectOfType(clr, TargetType);
    Console.WriteLine($"0x{obj.Address:x} - {obj.Type.Name}");
    var heap = clr.Heap;
    // Evaluation stack
    Stack<ulong> stack = new Stack<ulong>();

    HashSet<ulong> considered = new HashSet<ulong>();

    int count = 0;
    ulong size = 0;
    stack.Push(obj.Address);

    while (stack.Count > 0)
    {
        var objAddr = stack.Pop();
        if (considered.Contains(objAddr))
            continue;

        considered.Add(objAddr);

        ClrType type = heap.GetObjectType(objAddr);
        if (type == null)
        {
            continue;
        }
```

```
    count++;
    size += type.GetSize(objAddr);

    type.EnumerateRefsOfObject(objAddr,
                                delegate (ulong child,
                                           int offset)
    {
        if (child != 0 && !considered.Contains(child))
            stack.Push(child);
    });
}

Console.WriteLine($"Object Size: {obj.Size}");
Console.WriteLine($"Full size: {size}");
}
```

The output looks like this:

```
Object Size
=====
0x4636c24 - LargeMemoryUsage.D
Object Size: 12
Full size: 1000024
```

If you are interested only in aggregate object sizes, then PerfView can give you this information and allow you to aggregate sub-objects in multiple ways to get very fine-grained analysis. This was described in the previous section.

Which Objects Are Being Allocated On the LOH?

Understanding which objects are being allocated on the large object heap is critical to ensuring a well-performing system. The important rule discussed earlier in this chapter states that all objects should be cleaned up in a gen 0 collection, or they need to live forever.

Large objects are only cleaned up by an expensive gen 2 GC, so it violates that rule out of the gate.

To find out which objects are on the LOH, use PerfView and follow the previously given instructions for getting a GC event trace. In the resulting GC Heap Alloc Stacks view, in the By Name tab, you will find a special node that PerfView creates called “LargeObject.” Double-click on this to go to the Callers view, which shows which “callers” LargeObject has. In the sample program, they are all Int32 arrays. Double-clicking on those in turn will show where the allocations occurred.

Methods that call LargeObject

Name ?	Inc % ?	Inc ?
<input checked="" type="checkbox"/> LargeObject	8.8	302,824,200.0
+ <input checked="" type="checkbox"/> Type System.Int32[]	8.8	302,824,200.0
+ <input checked="" type="checkbox"/> OTHER <<clr!JIT_NewArr1>>	8.8	302,824,200.0
+ <input checked="" type="checkbox"/> AllocateAndRelease!AllocateAndRelease.Program.CreateArray(pool)	8.8	302,824,200.0
+ <input checked="" type="checkbox"/> AllocateAndRelease!AllocateAndRelease.Program.DoAnAllocation()	8.8	302,824,200.0
+ <input checked="" type="checkbox"/> AllocateAndRelease!AllocateAndRelease.Program.Main(class System.String)	8.8	302,824,200.0

Figure 2.25. PerfView can show large objects and their types with the stacks that allocated them.

CLR MD can also tell you which objects are in the large object heap.

```
private static void PrintLOHObjects(ClrRuntime clr)
{
    PrintHeader("LOH Objects (limit:10)");

    int objectCount = 0;
    const int MaxObjectCount = 10;
    if (clr.Heap.CanWalkHeap)
    {
        foreach (var segment in clr.Heap.Segments)
        {
            if (segment.IsLarge)
            {
                for (ulong objAddr = segment.FirstObject;
                     objAddr != 0;
                     objAddr = segment.NextObject(objAddr))
                {
                    var type = clr.Heap.GetObjectType(objAddr);

```

```
        if (type == null)
        {
            continue;
        }
        var obj = new ClrObject(objAddr, type);
        if (++objectCount > MaxObjectCount)
        {
            break;
        }
        Console.WriteLine(
            $"{obj.Address} {obj.Type.Name}");
    }
}
```

What Objects Are Being Pinned?

As covered earlier, a performance counter will tell you how many pinned objects the GC encounters during a collection, but that will not help you determine which objects are being pinned.

Use the Pinning sample project, which pins things via explicit `fixed` statements and by calling some Windows APIs.

Use WinDbg to view pinned objects with the !gchandles command:

```
0:010> !gchandles
Handle Type      Object    Size   Data Type
...
003511f8 Strong    01fa5dbc    52    System.Threading.Thread
003511fc Strong    01fa1330   112    System.AppDomain
003513ec Pinned    02fa33a8   8176   System.Object []
003513f0 Pinned    02fa2398   4096   System.Object []
003513f4 Pinned    02fa2178   528    System.Object []
003513f8 Pinned    01fa121c    12    System.Object
```

```
003513fc Pinned    02fa1020  4420      System.Object[]
003514fc AsyncPinned 01fa3d04     64
System.Threading.OverlappedData
```

You will usually see lots of `System.Object[]` objects pinned. The CLR uses these arrays internally for things like statics and other pinned objects. In the case above, you can see one `AsyncPinned` handle. This object is related to the `FileSystemWatcher` in the sample project.

Unfortunately, the debugger will not tell you why something is pinned, but often you can examine the pinned object and trace it back to the object that is responsible for it.

The following WinDbg session demonstrates tracing through object references to find higher-level objects that may give a clue to the origins of the pinned object. See if you can follow the trail of object references, starting with the address of the `AsyncPinned` handle from above.

```
0:010> !do 01fa3d04
Name:  System.Threading.OverlappedData
MethodTable: 64535470
EEClass:  646445e0
Size:  64(0x40) bytes
File:  C:\windows\Microsoft.Net\...\mscorlib.dll
Fields:
    MT  Field   Offset   Type VT   Attr   Value Name
64927254  4000700  4  System.IAsyncResult  0 instance 020a7a60
    m_asyncResult
64924904  4000701  8 ...ompletionCallback  0 instance 020a7a70
    m_iocb
...
0:010> !do 020a7a70
Name:  System.Threading.IOCompletionCallback
MethodTable: 64924904
EEClass:  6463d320
Size:  32(0x20) bytes
```

```
File: C:\windows\Microsoft.Net\...\mscorlib.dll
Fields:
    MT  Field   Offset   Type VT   Attr  Value Name
649326a4  400002d  4  System.Object  0 instance 01fa2bcc _target
...
0:010> !do 01fa2bcc
Name:  System.IO.FileSystemWatcher
MethodTable: 6a6b86c8
EEClass:  6a49c340
Size:  92(0x5c) bytes
File:  C:\windows\Microsoft.Net\...\System.dll
Fields:
    MT  Field   Offset   Type VT   Attr  Value Name
649326a4  400019a  4  System.Object  0 instance 00000000 __identity
6a699b44  40002d2  8 ...ponentModel.ISite  0 instance 00000000 site
...
```

While the debugger gives you the maximum power, it is cumbersome at best. Instead, you can use PerfView, which can simplify a lot of the drudgery.

With a PerfView trace, you will see a view called “Pinning at GC Time Stacks” that will show you stacks of the objects being pinned across the observed collections.

You can also approach pinning problems by looking at the free space holes created in the various heaps, which is covered in the next section.

Where Is Fragmentation Occuring?

Fragmentation occurs when there are freed blocks of memory inside segments containing used blocks of memory. Fragmentation can occur at multiple levels, inside a GC heap segment, or at the virtual memory level for the whole process. Fragmentation becomes a problem when there are so many small free blocks that they are not usable for future allocations.

Methods that call NonGen2

Name ?	
<input checked="" type="checkbox"/> NonGen2	
+ <input checked="" type="checkbox"/> Type System.Byte[]	
+ <input checked="" type="checkbox"/> LikelyAsyncDependentPinned	
+ <input checked="" type="checkbox"/> GC Location	
+ <input checked="" type="checkbox"/> Thread (6496) CPU=200ms	
+ <input checked="" type="checkbox"/> Process32 Pinning (7748)	
+ <input checked="" type="checkbox"/> ROOT	

Figure 2.26. PerfView will show you information about what types of objects are pinned across a GC, as well as some information about its likely origin.

Fragmentation in gen 0 is usually not an issue, unless you have a very severe pinning problem where you have pinned so many objects and each block of free space is too small to fulfill any new allocations. This will cause the size of the small object heap to grow and more garbage collections will occur.

Fragmentation is usually more of an issue in gen 2 or the large object heap, especially if you are not using background GC. You may see fragmentation rates that seem high, perhaps even 50%, but this is not necessarily an indication of a problem. Consider the size of the overall heap, and if it is acceptable and not growing over time, you probably do not need to take action.

First, you will want to know if fragmentation is happening at all. WinDbg can show you what percentage of a heap is free space, indicating fragmentation, using the !HeapStat command:

```
0:023> !HeapStat
Heap   Gen0    Gen1    Gen2    LOH
Heap0 2870384 2423640 93212392 9692760
```

Free space:	Percentage
Heap0 177940 21480 65552412 6324464 SOH: 66% LOH: 65%	

This prints each heap and tells you the percentage of free space in both small and large object heaps. For large object heap fragmentation, you can often deduce the likely culprits just by looking at which objects are on the large object heap and examining their sizes and related code. See earlier in this chapter for information on how to find this out.

You can get a summary of types and objects that are adjacent to free blocks with the !DumpHeap -stat command. At the very end of the heap summary, there will be some output like this:

Fragmented blocks larger than 0.5 MB:

Addr	Size	Followed by
16b61000	1.7MB	16d08948 System.Byte[]
16d08d7c	1.7MB	16ec4aa4 System.Byte[]
16f530c4	6.0MB	1755fb10 System.Byte[]
175e978c	0.6MB	17680ae0 System.Byte[]
176b9694	1.8MB	1787fff4 System.Byte[]
1e461000	1.5MB	1e5d7300 System.Byte[]
1e5d7734	1.4MB	1e74660c System.Byte[]
1e746a40	2.4MB	1e9a20d8 System.Byte[]

If you need detailed information about fragmentation, including which specific objects are causing the free space holes, you can use other WinDbg commands.

Get a list of free blocks with !DumpHeap -type Free:

```
0:010> !DumpHeap -type Free
Address      MT      Size
02371000 008209f8      10 Free
0237100c 008209f8      10 Free
02371018 008209f8      10 Free
023a1fe8 008209f8      10 Free
023a3fdc 008209f8      22 Free
023abdb4 008209f8     574 Free
023adfc4 008209f8      46 Free
023bbd38 008209f8     698 Free
```

```
023bdfe0 008209f8      18 Free
023d19c0 008209f8      1586 Free
023d3fd8 008209f8      26 Free
023e578c 008209f8      2150 Free
...
```

For each block, figure out which heap segment it is in with !eeheap -gc.

```
0:010> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x02371018
generation 1 starts at 0x0237100c
generation 2 starts at 0x02371000
ephemeral segment allocation context: none
    segment      begin      allocated      size
02370000 02371000 02539ff4 0x1c8ff4(1871860)
Large object heap starts at 0x03371000.
    segment      begin      allocated      size
03370000 03371000 03375398 0x4398(17304)
Total Size:      Size: 0x1cd38c (1889164) bytes.
-----
GC Heap Size:  Size: 0x1cd38c (1889164) bytes.
```

Dump all of the objects in that segment, or within a narrow range around the free space.

```
0:010> !DumpHeap 0x02371000 02539ff4
Address      MT      Size
02371000 008209f8      10 Free
0237100c 008209f8      10 Free
02371018 008209f8      10 Free
02371024 713622fc      84
02371078 71362450      84
```

023710cc	71362494	84
02371120	713624d8	84
02371174	7136251c	84
023711c8	7136251c	84
0237121c	71362554	12
...		

This is a manual and tedious process, but it does come in handy and you should understand how to do it. You can write scripts to process the output and generate the WinDbg commands for you based on previous output, but CLR Profiler can show you the same information in a graphical, aggregated manner that may be good enough for your needs.

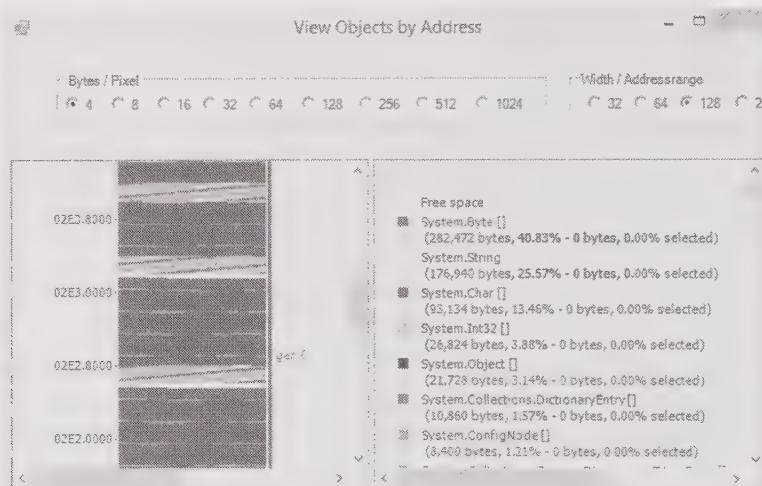


Figure 2.27. CLR Profiler can show you a visual representation of the heap that makes it possible to see what types of objects are next to free space blocks. In this image, the free space blocks are bordered by blocks of `System.Byte[]` and an assortment of other types.

PerfView can also tell you when fragmentation is occurring in the GCStats view. Look at the Frag % columns. However, it does not tell you why, exactly.

The CLR MD library allows you to build your own tool to highlight fragmentation for you. Each `ClrObject` has a `Type` property, which has an `IsFree` boolean property, which indicates if that type represents free space on the heap.

Virtual Memory Fragmentation

You may also get virtual memory fragmentation, which can cause an unmanaged allocation to fail because it cannot find a large enough range to satisfy the request. This can include allocating a new GC heap segment, which means your managed memory allocations will fail.

Use VMMap (part of SysInternals) to get a visual representation of your process. It will divide the heap into managed, native, and free regions. Selecting the Free portion will show you all Free segments. If the maximum size is insufficient for your requested memory allocation, you will get an `OutOfMemoryException`.

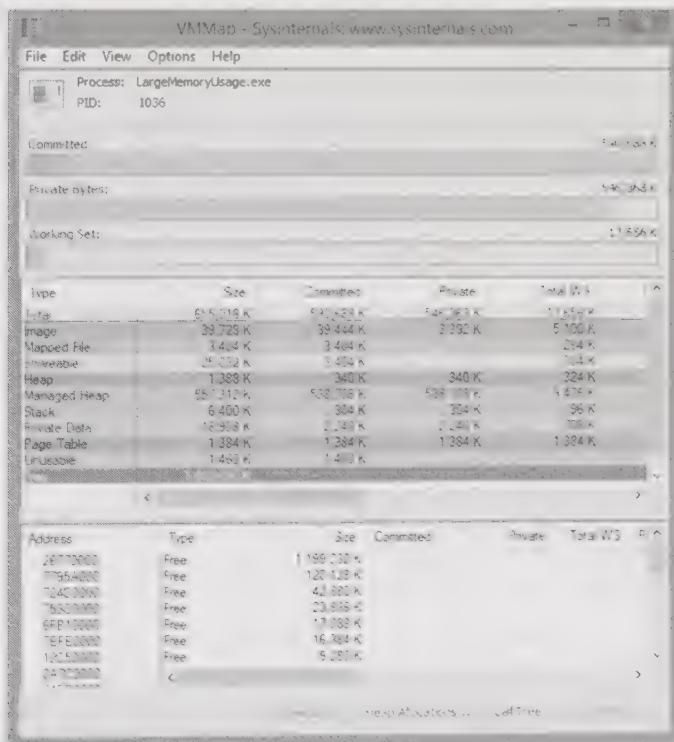


Figure 2.28. VMMap shows you tons of memory-related information, including the size of all free blocks in the address range. In this case, the largest block is over 1.1 GB in size—plenty!

VMMMap also has a fragmentation view that can show where these blocks fit in the overall process space.

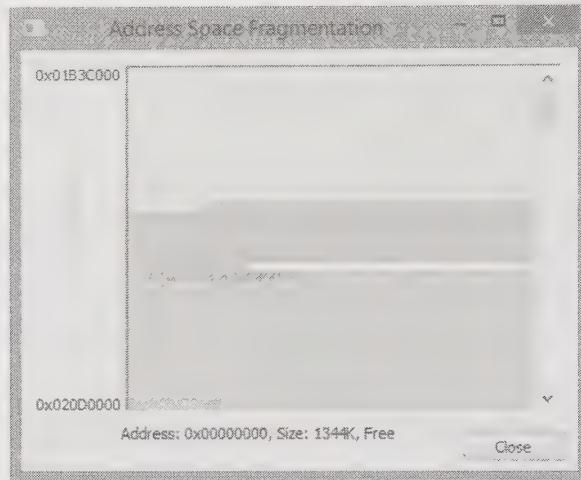


Figure 2.29. VMMMap's fragmentation view shows free space in the context of other segments.

You can also retrieve this information in WinDbg:

```
!address -summary
```

This command produces this output:

```
...
-- Largest Region by Usage -- Base Address -- Region Size --
Free           26770000    49320000 (1.144 Gb)
...
```

You can retrieve information about specific blocks with the command:

```
!address -f:Free
```

This produces output similar to:

BaseAddr	EndAddr+1	RgnSize	Type	State	Protect	Usage
0	150000	150000	MEM_FREE	PAGE_NOACCESS	Free	

Virtual memory fragmentation is more likely in 32-bit processes, where you are limited to just two gigabytes of address space for your program by default. The biggest symptom of this is an `OutOfMemoryException`. The easiest way to fix this is to convert your application to a 64-bit process, with its 128-terabyte address space. If you cannot do this, your only choice is to become far more efficient in memory allocations. You will need to compact the heaps and you may need to implement significant pooling.

What Generation Is An Object In?

You can retrieve this information from inside your app's own code by using the `GC.GetGeneration` method and passing it the object in question.

In WinDbg, once you obtain the address of the object of interest (say from `!DumpStackObjects` or `!DumpHeap`), use the `!gcwhere` command:

```
0:003> !gcwhere 02ed1fc0
Address      Gen Heap segment begin allocated size
02ed1fc0  1    0 02ed0000 02ed1000 02fe5d4c  0x14(20)
```

In CLR MD, you can use the `ClrHeap.GetGeneration` method:

```
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
}
```

Which Objects Survive Gen 0?

The simple way to do this is to simply enumerate all objects that are in the gen 1 or gen 2 portion of the heap.

CLR MD can do this for you with minimal code:

```
foreach(var obj in heap.EnumerateObjects())
{
    int gen = heap.GetGeneration(obj.Address);
    if (gen > 0)
    {
        // do some analysis
    }
}
```

On a big heap, it would be extremely inefficient to iterate through every object in the heap. If you are interested in just the gen 1 heap, for example, you can make this a little bit better by walking the heap per segment.

```
private static void PrintGen1ObjectsByHeapSegment(ClrRuntime clr)
{
    PrintHeader("Gen1 Objects by Heap Segment");
    if (clr.Heap.CanWalkHeap)
    {
        foreach(var segment in clr.Heap.Segments)
        {
            // Only the ephemeral segment contains gen0 and gen1
            if (segment.IsEphemeral)
            {
                //get range of gen 1
                ulong start = segment.Gen1Start;
                ulong end = start + segment.Gen1Length;
                Console.WriteLine(
                    $"Segment Info: Start: {start}, End {end}");
                for (ulong objAddr = segment.FirstObject;
                     objAddr != 0;
                     objAddr = segment.NextObject(objAddr))
                {
                    if (objAddr >= start && objAddr < end)
```

```
{  
    var type =  
        clr.Heap.GetObjectType(objAddr);  
    if (type == null)  
    {  
        continue;  
    }  
    var obj = new ClrObject(objAddr, type);  
    Console.WriteLine(  
        $"{obj.Address} {obj.Type.Name}");  
}  
}  
  
break;  
}  
}  
}  
}
```

On the other hand, perhaps you want to debug which objects survive a specific garbage collection – perhaps you are in the debugger, sitting at a breakpoint, and you want to know what happens after the next GC. This is possible in WinDbg, but it is fairly involved.

In WinDbg, execute these commands:

```
!FindRoots -gen 0  
g
```

This will set a breakpoint right before the next gen 0 collection begins. Once it breaks, you can send whatever commands you want to dump the objects on the heap. You can simply do:

```
!DumpHeap
```

This will dump every object on the heap, which may be excessive. Optionally, you can add the `-stat` parameter to limit output to a summary of the found objects (their counts, sizes, and types). However, if you want to limit your analysis to just gen 0, the `!DumpHeap` command allows you to specify an address range. Recall the description of memory segments from the top of the chapter and that gen 0 is at the end of the segment.



Figure 2.30. Basic segment layout

To get a list of heaps and segments, you can use the `eeheap -gc` command:

```
0:003> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x02ef0400
generation 1 starts at 0x02ed100c
generation 2 starts at 0x02ed1000
ephemeral segment allocation context: none
    segment begin allocated size
02ed0000 02ed1000 02fe5d4c 0x114d4c(1133900)
Large object heap starts at 0x03ed1000
    segment begin allocated size
03ed0000 03ed1000 041e2898 0x311898(3217560)
Total Size:           Size: 0x4265e4 (4351460) bytes.

-----
GC Heap Size:  Size: 0x4265e4 (4351460) bytes.
```

This command will give you a printout of each generation and each segment. The segment that contains gen 0 and gen 1 is called the ephemeral segment. `!eeheap` tells you the start of gen 0. To get the end of it, you merely need to find the segment that contains the start address. Each segment contains a number of addresses and the length. In the example above, the ephemeral segment starts at 02ed0000 and ends at 02fe5d4c. Therefore, the range of gen 0 on this heap is 02ef0400 - 02fe5d4c.

Now that you know this, you can put some limits on the !DumpHeap command and print only the objects in gen 0:

```
!DumpHeap 02ef0400 02fe5d4c
```

Once you have done that, you will want to compare what happens as soon as the GC is complete. This is a little trickier. You will need to set a breakpoint on an internal CLR method. This method is called when the CLR is ready to resume managed code. If you are using workstation GC, call:

```
bp clr!WKS::GCHeap::RestartEE
```

For server GC:

```
bp clr!SVR::GCHeap::RestartEE
```

Once you have set the breakpoints, continue execution (F5 or the g command). Once the GC is complete, the program will break again and you can repeat the !eeheap -gc and !DumpHeap commands.

Now you have two sets of outputs and you can compare them to see what changed and which objects are remaining after a GC. By using the other commands and techniques in this section, you can see who maintains a reference to that object.

Note

If you use server GC, then remember there will be multiple heaps. To do this kind of analysis, you will need to repeat the commands for each heap. The !eeheap command will print information for every heap in the process.

Who Is Calling GC.Collect Explicitly?

When code explicitly calls `GC.Collect`, it is called an “induced” garbage collection, and there are counters and ETW events that surface this information. However, they will not tell you who is calling it. You can easily search your own code base in Visual Studio or any advanced text editor, but if that turns up nothing, you will need to set a breakpoint on the `GC.Collect` method itself to see how your program gets to it.

In WinDbg, set a managed breakpoint on the `GC` class’s `Collect` method:

```
!bpmd mscorelib.dll System.GC.Collect
```

Continue executing. Once the breakpoint is hit, to see the stack trace of who called the explicit collection, do:

```
!DumpStack
```

What Weak References Are In My Process?

Because weak references are a type of GC handle, you can use the `!gchandles` command in WinDbg to find them:

```
0:003> !gchandles
Handle Type      Object  Size  Data Type
006b12f4 WeakShort  022a3c8c 100  System.Diagnostics.Tracing...
006b12fc WeakShort  022a3afc 52   System.Threading.Thread
006b10f8 WeakLong   022a3ddc 32   Microsoft.Win32.UnsafeNati...
006b11d0 Strong    022a3460 48   System.Object[]
...

```

Handles:

```
Strong Handles:    11
Pinned Handles:   5
Weak Long Handles: 1
Weak Short Handles: 2
```

Weak Short handles are the normal weak references you may use. Weak Long handles track whether a finalized object has been resurrected (objects without a finalizer always have short handles). Resurrection can occur when an object has been finalized, and rather than letting the GC clean it up, you decide to reuse it by assigning the object to a new reference from the finalizer. This can be relevant for pooling scenarios. However, it is possible to do pooling without finalization, and given the complexities of resurrection, just avoid this in favor of deterministic methods.

What Finalizable Objects Are On The Heap?

WinDbg's !FinalizeQueue command will show you all objects that are registered for finalization, as well as a summary of their types.

```
0:042> !FinalizeQueue
SyncBlocks to be cleaned up: 0
Free-Threaded Interfaces to be released: 0
MTA Interfaces to be released: 0
STA Interfaces to be released: 0
-----
generation 0 has 13 finalizable objects (288603b4->288603e8)
generation 1 has 6 finalizable objects (2886039c->288603b4)
generation 2 has 57247 finalizable objects (28828520->2886039c)
Ready for finalization 0 objects (288603e8->288603e8)
Statistics for all finalizable objects
(including all objects ready for finalization):
    MT Count TotalSize Class Name
72753184      1        12 System.WeakReference`1...
6df6bea8      1        12 System.Windows.Forms.VisualStyles...
6df68c44      1        12 System.Windows.Forms.ImageList...
584582f0      1        12 System.WeakReference`1...
58443158      1        12 Microsoft.Build.BackEnd.Components...
...
```

If you want to see a summary of objects that are ready for finalization, you can execute:

```
!FinalizeQueue -detail
```

This will show you a list of type names that are currently “freachable” (that is, eligible to have their finalizers called). If you want to get the specific objects that are in that category, you can use the address range given in the output to dump all objects within the “ready for finalization” range:

```
!DumpHeap 288603e8 288606c4
```

In CLR MD, you can use the `EnumerateFinalizableObjectAddresses` method to enumerate all finalizable objects:

```
private static void PrintFinalizableObjects(ClrRuntime clr)
{
    foreach (var objAddr in
        clr.Heap.EnumerateFinalizableObjectAddresses())
    {
        ClrType type = clr.Heap.GetObjectType(objAddr);
        if (type == null)
        {
            continue;
        }
        ClrObject obj = new ClrObject(objAddr, type);
        // Do something with the object...
    }
}
```

Unfortunately, this does not tell you whether those objects are ready for finalization.

Summary

You need to understand garbage collection in-depth to truly optimize your applications. Choose the right configuration settings for your application, such as server GC if your application is the only one running on the machine, but be

wary of using advanced configuration settings. Ensure that object lifetime is short, allocation rates are low, and that any objects that must live longer than the average GC frequency are pooled or otherwise kept alive in gen 2 forever. Make judicious use of `stackalloc` to avoid heap allocations.

Avoid pinning and finalizers if possible. Any LOH allocations should be pooled and kept around forever to avoid full GCs. Reduce LOH fragmentation by keeping objects a uniform size and occasionally compacting the heap on-demand. Consider GC notifications to avoid having full collections impact application processing at inopportune times.

The garbage collector is a deterministic component and you can control its operation by closely managing your object allocation rate and lifetime. You are not giving up control by adopting .NET's garbage collector, but it does require a little more subtlety and analysis.

Chapter 3

JIT Compilation

.NET code is distributed as assemblies of Microsoft Intermediate Language (MSIL, or just IL for short). This language is somewhat assembly-like, but simpler. If you wish to learn more about IL or other CLR standards, do an Internet search for “ECMA C# CLI standards.”

When your managed program executes, it loads the CLR which will start executing some wrapper code. This is all machine code. The first time a managed method is called from your assembly, it actually runs a stub that executes the Just-in-Time (JIT) compiler which will convert the IL for that method to the hardware’s machine instructions. This process is called just-in-time compilation (“JITting”). The stub is replaced and the next time that method is called, the assembly instructions are called directly. This means, the first time any method is called, there is always a performance hit. In most cases, this hit is small and can be ignored. Every time after that, the code executes directly and incurs no overhead.

While all code in a method will be converted to assembly instructions when JITted, some pieces may be placed into “cold” code sections of memory, separate from the method’s normal execution path. These rarely executed paths will thus not push out other code from the “warm” sections, allowing for better overall performance as the commonly executed code is kept in memory, but the cold pages may be paged out. For this reason, rarely used things like error and exception-handling paths can be expensive.

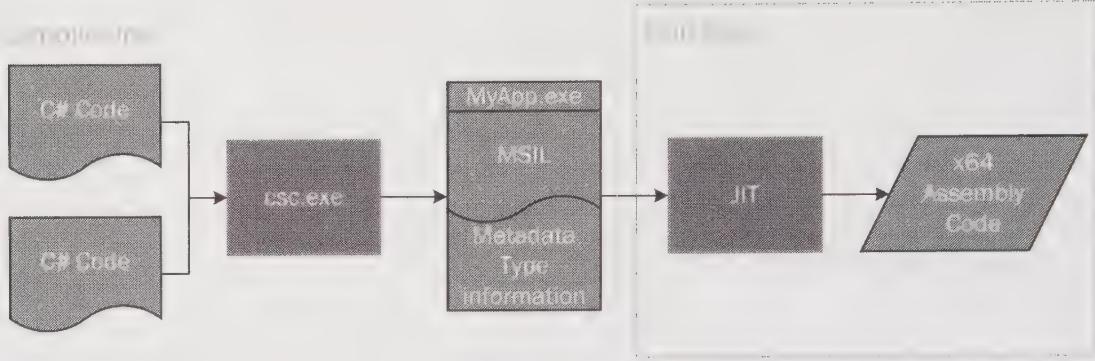


Figure 3.1. Compilation and JITting flow.

In most cases, a method will only need to be JITted once. The exception is when the method has generic type arguments. If any of the type arguments are value types, then new code must be generated for each type. If all the type parameters are reference types, then only one copy of the native code is generated because, despite the different types, in machine code, each reference just looks like a standard pointer (4 or 8 bytes). Note that this does not mean the fundamental types are the same—the type system still maintains integrity and, for example, `List<string>` is still distinct from `List<Regex>`. It is just that the underlying machine code implementation of the methods does not need to care about those distinctions, so it does not generate duplicate code.

You need to be concerned about JIT costs if this first-time warm-up cost is important to your application or its users. Most applications only care about steady-state performance, but if you must have extremely high-availability, JIT can be an issue that you will need to optimize for. In this case, NGEN may be the right solution for you. This chapter will tell you how to take advantage of it, or if it is even the right thing.

Benefits of JIT Compilation

Code that is just-in-time compiled has some significant advantages over compiled unmanaged code.

1. Good Locality of Reference: Code that is used together will often be in the same page of memory or processor cache line, preventing expensive page faults and relatively expensive main memory accesses.
2. Potentially Reduced Memory Usage: There is CLR overhead for managed DLLs and their metadata, but it only compiles those methods that are actually used.
3. Cross-assembly Inlining: Methods from other DLLs, including the .NET Framework, can be inlined into your own application, which can be a significant time savings at runtime.

There is also a benefit of hardware-specific optimizations, but in practice there are only a few actual optimizations for specific platforms. However, it is becoming increasingly possible to target multiple platforms with the same code, and perhaps we will see more aggressive platform-specific optimizations in the future.

Most code optimizations in .NET do not take place in the language compiler (the transformation from C#/VB.NET to IL); rather, they occur on-the-fly in the JIT compiler.

JIT in Action

You can easily see the IL-to-assembly-code transformation in action. As a simple example, here is the JitCall sample program that demonstrates the code fix-up that JIT does behind the scenes:

```
static void Main(string[] args)
{
    int val = A();
    int val2 = A();
    Console.WriteLine(val + val2);
}

[MethodImpl(MethodImplOptions.NoInlining)]
```

```
static int A()
{
    return 42;
}
```

This method is decorated with the `MethodImplOptions.NoInlining` attribute. The reason for this is to force the method call to remain, even in optimized code. If it is not present, the call gets optimized out completely, as does the addition, leaving you with a constant value being put into the argument for the `WriteLine` method:

```
029D0450  mov          ecx, 54h
029D0455  call         72B2CE9C
029D045A  ret
```

We want to look at the JIT behavior of a simple method, so applying the attribute `MethodImplOptions.NoInlining` is a simple way of ensuring that the method sticks around for us to analyze.

To see what happens, first get the disassembly of `Main`. Getting to this point is a little bit of a trick. First, we need to launch the program and interrupt it before `Main` executes.

1. Launch WinDbg.
2. File | Open Executable... (Ctrl+E).
3. Navigate to the JitCall binary. Make sure you pick the Release version of the binary or the assembly code will look quite different than what is printed here.
4. The debugger will immediately break.
5. Run the command: `sxe ld clrjit`. This will cause the debugger to break when `clrjit.dll` is loaded. This is convenient because once this is loaded you can set a breakpoint on the `Main` method before it is executed.
6. Run the command: `g`.

7. The program will execute until clrjit.dll is loaded and you see output similar to the following:

```
ModLoad: 6fe50000 6fec0d000
C:\Windows\Microsoft.NET\Framework\v4.0.30319\clrjit.dll
```

Next, we'll break into Main:

1. Run the command: `.loadby sos clr`.
2. Run the command: `!bpmd JitCall Program.Main`. This sets the breakpoint at the beginning of the `Main` function.
3. Run the command: `g`.
4. WinDbg will break right inside the `Main` method. You should see output similar to this:

```
(11b4.10f4): CLR notification exception
- code e0444143 (first chance)
JITTED JitCall!JitCall.Program.Main(System.String[])
Setting breakpoint: bp 007A0050
[JitCall.Program.Main(System.String[])]
Breakpoint 0 hit
```

Finally, we are in the right place. Open the Disassembly window (Alt+7). You may also find the Registers window interesting (Alt+4).

The disassembly of `Main` looks like this:

```
push    ebp
mov     ebp,esp
push    edi
push    esi

; Call A
call    dword ptr ds:[0E537B0h]  ds:002b:00e537b0=00e5c015
```

```
mov    edi,eax
call   dword ptr ds:[0E537B0h]
mov    esi,eax

call   mscorelib_ni+0x340258 (712c0258)
mov    ecx,eax
add    edi,esi
mov    edx,edi
mov    eax,dword ptr [ecx]
mov    eax,dword ptr [eax+38h]

; Call Console.WriteLine
call   dword ptr [eax+14h]
pop    esi
pop    edi
pop    ebp
ret
```

There are two calls to the same pointer (the specific values you see will differ). This is the function call to A. Set break points on both of these lines and start stepping through the code one instruction at a time, making sure to step into the calls. The pointer at 0E537B0h will get updated after the first call.

Stepping into the first call to A, you can see that it is little more than a `jmp` to the CLR method `ThePreStub`. There is no return from this method here because `ThePreStub` will do the return.

```
mov    al,3
jmp   00e5c01d
mov    al,6
jmp   00e5c01d
(00e5c01d) movzx   eax,al
shl    eax,2
add    eax,0E5379Ch
jmp   clr!ThePreStub (72102af6)
```

On the second call to A, you can see that the function address of the original pointer was updated and the code at the new location looks more like a real method. Notice the 2Ah (our decimal 42 constant value from the source) being assigned and returned via the eax register.

```

012e0090 55          push   ebp
012e0091 8bec        mov    ebp,esp
012e0093 b82a000000  mov    eax,2Ah
012e0098 5d          pop    ebp
012e0099 c3          ret

```

For most applications, this first-time, or warm-up, cost is not significant, but there are certain types of code that lend themselves to high JIT time, which we will examine in the next few sections.

JIT Optimizations

The JIT compiler will perform some standard optimizations such as method inlining and array range check elimination, but there are things that you should be aware of that can prevent the JIT compiler from optimizing your code. Some of these topics have their own treatments in Chapter 5. Note that because the JIT compiler executes during runtime, it is limited in how much time it can spend doing optimizations. Despite this, it can do many important improvements.

One of the biggest classes of optimizations is method inlining, which puts the code from the method body into the call site, avoiding a method call in the first place. Inlining is critical for small methods which are called frequently, where the overhead of a function call is larger than the function's own code.

All of these things prevent inlining:

- Virtual methods.
- Interfaces with diverse implementations in a single call site. See Chapter 5 for a discussion of the interface dispatch problem.
- Loops.
- Exception handling.

- Recursion.
- Method bodies larger than 32 bytes of IL. You can use an IL analysis tool (discussed in Chapter 1) to view the size of methods.

As of .Net 4.6, a new version of the JIT was released. Formerly known as RyuJIT, it featured significantly improved code generation performance as well as improved generated code quality, particularly for 64-bit code.

Be careful of calling properties or methods inside loops. In most cases, the JIT cannot optimize these calls out. You should take care to make loop bodies as cheap as possible and do these optimizations yourself. If a method or property can be called outside of a loop, then it usually should, with the result being stored in a local variable.

Reducing JIT and Startup Time

The other major factor in considering the JIT compiler is the amount of time it takes to generate the code. This mostly comes down to a factor of how much code needs to be JITted.

For example, large methods will take longer to JIT than shorter methods. If you have a single large method with a lot of branching in it, you still pay the JIT cost even if most of the method never executes. Breaking it up into smaller methods may help your up front JIT cost.

There are also language features and APIs that cause a large amount of code to be generated. In particular, be aware of the following situations:

- LINQ
- The `dynamic` keyword
- `async` and `await`
- Regular expressions

- Code generation
- Many types of serializers

All of these have a simple fact in common: Much more code is possibly hidden from you and actually executed than is obvious from your source. All of that hidden code may require significant time to be JITted. With regular expressions and generated code in particular, there is likely to be a pattern of large, repetitive blocks of code.

While code generation is usually something you would write for your own purposes, there are some areas of the .NET Framework that will do this for you, the most common being regular expressions and XML serialization. Before execution, regular expressions can be converted to an IL state machine in a dynamic assembly and then JITted. This takes more time up front, but saves a lot of time with repeated execution (as long as you make the regular expression static). You usually want to enable this option, but you probably want to defer it until it is needed so that the extra compilation does not impact application start time. Regular expressions can also trigger some complex algorithms in the JIT that take longer than normal, most of which are improved in the JIT that ships in .NET 4.6. As with everything else in the book, the only way to know for sure is to measure. See Chapter 6 for more discussion of regular expressions.

Even though code generation is implicated as a potential exacerbation of JIT challenges, as we will see in Chapter 5, code generation in a different context can get you out of some other performance issues.

LINQ's syntactic simplicity can belie the amount of code that actually runs for each query. It can also hide things like delegate creation, memory allocations, and more. Simple LINQ queries may be OK, but as in most things, you should definitely measure.

The primary issue with `dynamic` code is, again, the sheer amount of code that it translates to. Jump to Chapter 5 to see what `dynamic` code looks like under the covers.

There are other factors besides JIT, such as I/O, that can increase your warm-up costs, and it behooves you to do an accurate investigation before assuming JIT is the only issue. Each assembly has a cost in terms of disk access for reading the file, internal overhead in the CLR data structures, and type loading.

You may be able to reduce some of the I/O cost by combining many small assemblies into one large one, but type loading is likely to consume as much time as JITting.

If you do have a lot of JIT happening, you should see stacks containing calls to PreStubWorker and other methods that eventually end up inside clrjit.dll.

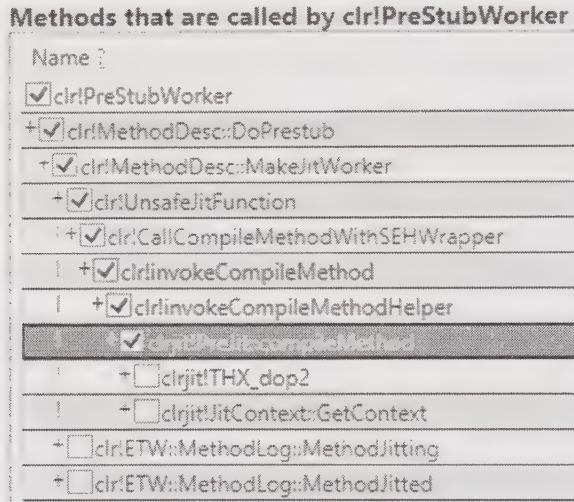


Figure 3.2. PerfView's CPU profiling will show you any JIT stubs that are being called.

Later in this chapter, we will see how PerfView can show you exactly which methods are being JITted and how long each one took.

Optimizing JITting with Profiling (Multicore JIT)

.NET 4.5 added an API that tells .NET to profile your application's startup and store the results on disk for future references. On subsequent startups, this profile is used to start generating the assembly code before it is executed. This happens in a dedicated thread apart from your own code's threads (and giving this feature the nickname Multicore JIT). The saved profiles allow this generated code to have all the same benefits of locality as JITting. The profiles are updated automatically on each execution of your program.

To use it, simply call this at the beginning of your program:

```
ProfileOptimization.SetProfileRoot(@"C:\MyAppProfile");
ProfileOptimization.StartProfile("default");
```

Note that the profile root folder must already exist, and you can name your profiles, which is useful if your app has different modes with substantially different execution profiles.

If you do use this feature, keep this in mind while doing your own profiling of startup performance: the applied optimizations will change your measurements. Depending on what you are looking for, you may want to temporarily disable multicore JIT.

When to Use NGEN

If application startup or warm-up costs are too high and the profile optimization mentioned earlier does not satisfy your performance requirements, then NGEN may be appropriate.

NGEN stands for Native Image Generator. It works by converting your IL assembly to a native image – in effect, running the JIT compiler and saving the results to a native image assembly cache. This results in faster startup time and less JIT overall. This native image should not be confused with native code in the sense of unmanaged code. Despite the fact that the image is now mostly assembly language, it is still a managed assembly because it must run under the CLR.

If your original assembly is called foo.dll, NGEN will generate a file called foo.ni.dll and put it in the native image cache. Whenever foo.dll is requested to be loaded, the CLR will inspect the cache for a matching .ni. file and verify that it matches the IL exactly. It does this using a combination of time stamps, names, and GUIDs to 100% ensure that it is the correct file to load.

NGEN has its place, but it does have some disadvantages. The first is that you lose locality of reference, as all the code in an assembly is placed sequentially, regardless of how it is actually executed. In addition, you can lose certain op-

timizations such as cross-assembly inlining. You can get most of these optimizations back if all of the assemblies are available to NGEN at the same time. You must also update the native images every time there is a change – not a big deal, but an extra step for deployment. NGEN can be very slow and native images can be significantly larger than their managed counterparts. Sometimes, JIT will produce more optimized code, especially for commonly executed paths. Before deciding to use NGEN, remember the prime directive of performance: **Measure, Measure, Measure!** See the tips at the end of this chapter for how to measure JIT costs in your application.

Most usages of generics can be successfully NGENed, but there are cases where the compiler cannot figure out the right generic types ahead of time. This code will still be JITted at runtime. And of course, any time you rely on dynamic type loading or generation, those pieces cannot be NGENed ahead of time.

To NGEN an assembly from the command line, execute this command:

```
D:\>ngen install ReflectionExe.exe
```

```
1> Compiling assembly D:\...\ReflectionExe.exe (CLR v4.0.30319) ...
2> Compiling assembly ReflectionInterface, Version=1.0.0.0,
   Culture=neutral, PublicKeyToken=null (CLR v4.0.30319) ...
```

From the output you see that there are actually two files being processed. NGEN will automatically look in the target file's directory and NGEN any dependencies it finds. It does this by default to allow the code to make cross-assembly calls in an efficient away (such as inlining small methods). You can suppress this behavior with the **/NoDependencies** flag, but there may be a significant performance hit during runtime.

To remove an assembly's native image from the machine's native image cache, you can run:

```
D:\>ngen uninstall ReflectionExe.exe
```

```
Uninstalling assembly D:\...\ReflectionExe.exe
```

You can verify that a native image was created by displaying the native image cache:

```
D:\>ngen display ReflectionExe
```

NGEN Roots:

```
D:\Book\ReflectionExe\bin\Release\ReflectionExe.exe
```

NGEN Roots that depend on "ReflectionExe":

```
D:\Book\ReflectionExe\bin\Release\ReflectionExe.exe
```

Native Images:

```
ReflectionExe, Version=1.0.0.0, Culture=neutral,
```

```
PublicKeyToken=null
```

You can also display all cached native images by running the command:

```
ngen display.
```

Note that a native image file is always larger than the purely managed IL version of it. The IL version is contained wholly inside the native image, and in addition, x86/x64 code can be more verbose than IL. For example, for the ReflectionExe.exe file used above, its file size is 5,632 bytes, while the native image in the GAC is 11,264 bytes. I have sometimes seen as much as a 4x file size increase, depending on the amount of metadata and type of code present in the managed assemblies.

Optimizing NGEN Images

I said above that one of the things you lose with NGEN is locality of reference. Starting with .NET 4.5, you can use a tool called Managed Profile Guided Optimization (MPGO) to fix this problem to a large extent. Similar to Profile Optimization for JIT, this is a tool that you manually run to profile your application's startup (or whatever scenario you want). NGEN will then use the profile to create a native image that is better optimized for the common function chains.

MPGO is included with Visual Studio 2012 and higher. To use, run the command:

```
Mpgo.exe -scenario MyApp.exe -assemblyList *.* -OutDir c:\Optimized
```

This will cause MPGO to run on some framework assemblies and then it will execute MyApp.exe. Now the application is in training mode. You should exercise the application appropriately and then shut it down. This will cause a new, optimized assembly to be created in the C:\Optimized directory.

To take advantage of the optimized assembly, you must run NGEN on it:

```
Ngen.exe install C:\Optimized\MyApp.exe
```

This will create optimized images in the native image cache. Next time the application is run, these new images will be used.

To use the MPGO tool effectively, you will need to incorporate it into your build system so that its output is what gets shipped with your application.

.NET Native

If you are building Universal Windows Platform applications, then you can use .NET Native, a compiler that transforms your compiled managed application into native code, similar to NGEN, but with these advantages:

- A newer compiler, based on the Visual C++ native compiler.
- Self-contained applications. There is no CLR dependency, once compiled. The CLR is reduced to a single DLL that ships with your app. Any framework code that you actually use is statically linked inside your executable.

The compiler creates the CLR-in-a-DLL by running a dependency reducer engine on your code. This process is informally known as “tree shaking.” It analyzes your code, configuration, XAML files, type arguments, and more to determine everything that could possibly run. This produces fast, compact applications that start up with very little lag.

There are, however, a number of downsides, mostly driven by the requirement that no JITting is allowed:

- No reflection
- No dynamic assembly loading or code invocation
- No serialization or deserialization
- No COM interop (regular P/Invoke is OK)
- Only works for Universal Windows Platform apps now

To use .NET Native, you need to create a new Universal Windows Platform application in Visual Studio. Release builds will automatically build with .NET Native.

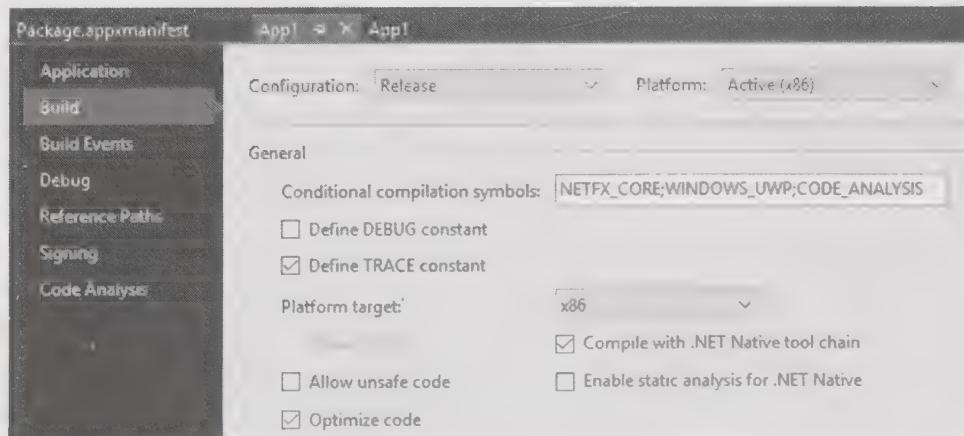


Figure 3.3. Universal Windows Platform applications allow you to specify whether .NET Native is used. It is on by default for release builds.

Custom Warmup

If the other techniques mentioned in this chapter do not work, you can take matters into your own hands and implement a system to warmup your code by executing it before it is executed in a production scenario. Just executing each method will cause the JIT to happen. This is a popular method in online scenarios where wall-clock time is vitally important. You may have internal timeouts of a few milliseconds, or a client that gives up after a few seconds. If you have a large code base, the first request (or first many requests) may entirely fail due to JITting.

If you have a significant amount of JIT, profile-guided optimization may not be good enough. You may just need to exercise the code in a test or offline mode before it handles a real workload.

Before deciding this is the right approach, it would be good to ask yourself some guiding questions:

1. Is your code designed in such a way that it can easily be called in a warmup scenario?
2. How long does warmup take? Can your application afford to take longer to startup?
3. If you have a lot of code to warmup, can you JIT in parallel?
4. If you need data to warmup, can this be generated automatically?
5. Does your warmup code replay safe data (i.e., non-production, non-customer, replayable data)?
6. Does your warmup code impact other systems? Even if your application can handle 32 cores of solid JITting, will this impact external systems in a negative manner?
7. Will warming up code cause metrics to be impacted? Minimize these, split them out, or otherwise tune metrics to exclude warmup data.

The answer to some of these questions may need to be some prototyping. Doing warmup does not have to be a huge feature, but it is likely to be non-trivial.

Warmup is unlikely to cause every single piece of necessary code to JIT, but if it gets to a significant percentage (enough to avoid errors or timeouts later), then it can be good enough.

When JIT Cannot Compete

The JIT is great for most applications. If there are performance problems, there are usually bigger issues than pure code generation quality or speed. However, there are some areas where the JIT has room to improve.

One major situation where the JIT compiler is not going to be quite as good as a native code compiler is with direct native memory access vs. managed array access. For one, accessing native memory directly usually means you can avoid the memory copy that will come with marshaling it to managed code. While there are ways around this with things like `UnmanagedMemoryStream`, which will wrap a native buffer inside a `Stream`, you are really just making an unsafe memory access.

If you do transfer the bytes to a managed buffer, the code that accesses the buffer will have boundary checks. In many cases, these checks can be optimized away, but it is not guaranteed. With managed buffers, you can wrap a pointer around them and do some unsafe access to get around some of these checks.

If you find that unmanaged code really is more efficient at this kind of processing, you can try marshaling the entire data set to a native function via P/Invoke, compute the results with a highly optimized C++ DLL, and then return the results back to managed code. You will have to profile it to see if the data transfer cost is worth it.

Mature C++ compilers may also be better at other types of optimizations such as inlining or optimal register usage, but this is more likely to change with future versions of the JIT compiler.

With applications that do an extreme amount of array or matrix manipulation, you will have to consider this trade-off between performance and safety. For most applications, frankly, you will not have to care and the boundary checks are not a significant overhead. However, if you are doing significant mathematical manipulation, one possible option is to make explicit use of Single Instruction, Multiple Data (SIMD) instructions, which became available to the JIT compiler in .NET 4.6. See Chapter 6 for examples of how to use these.

Investigating JIT Behavior

Performance Counters

The CLR publishes a number of counters in the .NET CLR Jit category, including:

- # of IL Bytes Jitted
- # of Methods Jitted
- % Time in Jit
- IL Bytes Jitted /sec
- Standard Jit Failures
- Total # of IL Bytes Jitted (exactly the same as “# of IL Bytes Jitted”)

Those are all fairly self-explanatory except Standard Jit Failures. Failures can occur only if the IL is unverified or there is an internal JIT error.

Closely related to JITting, there is also a category for loading, called .NET CLR Loading. A few of them are:

- % Time Loading

- Bytes in Loader Heap
- Total Assemblies
- Total Classes Loaded

ETW Events

With ETW events, you can get extremely detailed performance information on every single method that gets JITted in your process, including the IL size, native size, and the amount of time it took to JIT.

- **MethodJittingStarted:** method is being JIT-compiled. Fields include:
 - MethodID: Unique ID for this method.
 - ModuleID: Unique ID for the module to which this method belongs.
 - MethodILSize: Size of the method's IL.
 - MethodNameSpace: Full class name to which this method belongs.
 - MethodName: Name of the method.
 - MethodSignature: Comma-separated list of type names from the method signature.
- **MethodLoad V1:** A method is done JITting and has been loaded. Generic and dynamic methods do not use this version. Fields include:
 - MethodID: Unique ID for this method.
 - ModuleID: Unique ID for this module to which this method belongs.
 - MethodSize: Size of the compiled assembly code after JIT.
 - MethodStartAddress: Start address of the method.
 - MethodFlags:
 - 0x1: Dynamic method
 - 0x2: Generic method

- 0x4: JIT-compiled (if missing, it was NGENed)
 - 0x8: Helper method
- **MethodLoadVerbose V1:** A generic or dynamic method has been JITted and loaded.
 - It has most of the same fields as MethodLoad_V1 and MethodJittingStarted.

What Code Is Jitted?

If you need to audit the code in your process, perhaps to see which assembly uses the most memory after JIT, you will need to examine the IL and native code sizes of all the methods in the process.

Using CLR MD, you can analyze every method in the process, seeing how large the IL is, as well as the amount of native code JITted from the IL. Here is a method that prints the top 10 largest methods in a process:

```
class MethodSize
{
    public string Module { get; set; }
    public string TypeName { get; set; }
    public string Name { get; set; }
    public ulong ILSize { get; set; }
    public ulong NativeSize { get; set; }
}

const string TargetProcessName = "LargeMemoryUsage.exe";

private static void PrintTop10BiggestMethods(ClrRuntime clr)
{
    PrintHeader("Top 10 Methods");
    List<MethodSize> methods = new List<MethodSize>();

    for (int i = 0; i < clr.Modules.Count; i++)
    {
        // Only look at our own methods
        var module = clr.Modules[i];
```

```
if (!module.FileName.EndsWith(TargetProcessName))
{
    continue;
}
string filename = Path.GetFileName(module.FileName);

foreach (var type in module.EnumerateTypes())
{
    for (var iMethod = 0;
        iMethod < type.Methods.Count;
        iMethod++)
    {
        ulong ilSize = 0;
        ulong nativeSize = 0;

        var method = type.Methods[iMethod];

        if (method.IL != null)
        {
            ilSize += (ulong)method.IL.Length;

            if (method.ILOffsetMap != null)
            {
                for (var iOffset = 0;
                    iOffset < method.ILOffsetMap.Length;
                    iOffset++)
                {
                    var entry = method.ILOffsetMap[iOffset];
                    var size = entry.EndAddress -
                               entry.StartAddress;
                    nativeSize += size;
                }
            }
        }
        var methodSize = new MethodSize()
        {
            Module = filename,
            TypeName = type.Name,
            Name = method.Name,
            ILSize = ilSize,
            NativeSize = nativeSize
        };
        methods.Add(methodSize);
    }
}
```

```

        }
    }

    methods.Sort((a, b) =>
{
    return -a.NativeSize.CompareTo(b.NativeSize);
});

Console.WriteLine(
    "Module, Type, Method, IL Size, Native Size");
Console.WriteLine(
    "-----");
for (int i=0;i<Math.Min(10, methods.Count);i++)
{
    var method = methods[i];
    Console.WriteLine(
        $"{method.Module}, {method.TypeName}, {method.Name}, " +
        $"{method.ILSize}, {method.NativeSize}");
}
}
}

```

This produces output similar to:

```

Top 10 Methods
=====
Module, Type, Method, IL Size, Native Size
-----
LargeMemoryUsage.exe, LargeMemoryUsage.Program, Main, 116, 348
LargeMemoryUsage.exe, LargeMemoryUsage.Program, GetNewObject, 67, 250
LargeMemoryUsage.exe, LargeMemoryUsage.Base, .ctor, 21, 113
LargeMemoryUsage.exe, LargeMemoryUsage.Program, .cctor, 16, 88
LargeMemoryUsage.exe, LargeMemoryUsage.C, .ctor, 14, 70
LargeMemoryUsage.exe, LargeMemoryUsage.D, .ctor, 14, 69
LargeMemoryUsage.exe, LargeMemoryUsage.B, .ctor, 14, 69
LargeMemoryUsage.exe, LargeMemoryUsage.A, .ctor, 14, 69
LargeMemoryUsage.exe, LargeMemoryUsage.D, ToString, 12, 51
LargeMemoryUsage.exe, LargeMemoryUsage.C, ToString, 12, 51

```

What Methods and Modules Take the Longest To JIT?

In general, JIT time is directly proportional to the amount of IL instructions in a method, but this is complicated by the fact that type loading time can also be included in this time, especially the first time a module is used. Some patterns can also trigger complex algorithms in the JIT compiler, which may run longer. You can use PerfView to get very detailed information about JITting activity in your process. If you collect the standard .NET events, you will get a special view called “JITStats.” Here is some of the output from running it on the PerfCountersTypingSpeed sample project:

Name	JitTime msec	Num Methods	IL Size	Native Size
PerfCountersTypingSpeed.exe	12.9	8	1,756	3,156

JitTime msec	IL Size	Native Size	Method Name
9.7	22	45	PerfCountersTypingSpeed.Program.Main()
0.3	176	313	PerfCountersTypingSpeed.Form1.ctor()
1.4	1,236	2,178	PerfCountersTypingSpeed.Form1.InitializeComponent()
0.8	107	257	PerfCountersTypingSpeed.Form1.CreateCustomCategories()
0.3	143	257	PerfCountersTypingSpeed.Form1.timer_Tick(class System.Object,class System.EventArgs)
0.1	23	27	PerfCountersTypingSpeed.Form1.OnKeyPress(class System.Object,class System.Windows.Forms.KeyPressEventArgs)
0.2	19	36	PerfCountersTypingSpeed.Form1.OnClosing(class System.ComponentModel.CancelEventArgs)
0.1	30	43	PerfCountersTypingSpeed.Form1.Dispose(bool)

The only method that takes more time to JIT than its IL size would suggest is `Main`, which makes sense because this is where you will pay for more loading costs.

Examine JITted code

In WinDbg or Visual Studio, you can easily see the disassembled code around the current instruction location, and from there jump to anywhere else as well. In Visual Studio, when you are at a debug break point, right-click anywhere in the source and select “Go to disassembly.”

You can also easily get an annotated dump of the disassembled code directly of a specific method in WinDbg using the `!U` command. To do this, you need to get the `MethodDesc` structure pointer.

```
0:000> !DumpStack
OS Thread Id: 0x5580 (0)
Current frame: ntdll!NtDeviceIoControlFile+0xc
ChildEBP RetAddr Caller, Callee
...
012ff2e4 7217c50a (MethodDesc 716f0d54 +0xe6
    System.Console.ReadKey(Boolean)), calling 71995a48
012ff374 039b0514 (MethodDesc 035d4d64 +0x9c
    LargeMemoryUsage.Program.Main(System.String[])),
    calling (MethodDesc 716f0d54 +0 System.Console.ReadKey(Boolean))
012ff398 72cceeb16 clr!CallDescrWorkerInternal+0x34
...
```

From this, we will use the `MethodDesc` value for the `Main` method, `035d4d64`:

```
0:000> !U 035d4d64
Normal JIT generated code
LargeMemoryUsage.Program.Main(System.String[])
Begin 039b0478, size bc
```

D:\...\LargeMemoryUsage\Program.cs @ 16:

```
039b0478 55          push    ebp
039b0479 8bec        mov     ebp,esp
039b047b 57          push    edi
039b047c 56          push    esi
039b047d 53          push    ebx
039b047e 83ec10      sub     esp,10h
039b0481 b9926a6371  mov     ecx,offset
                         mscorlib_ni!System.Collections.IStructuralEquatable.Equals+0x99
                         (71636a92)
039b0486 bae8030000  mov     edx,3E8h
039b048b e8202dc1ff  call    035c31b0
                         (JitHelp: CORINFO_HELP_NEWARR_1_OBJ)
039b0490 8945e4      mov     dword ptr [ebp-1Ch],eax
```

D:\...\LargeMemoryUsage\Program.cs @ 18:

```
039b0493 b9dc7eb471  mov     ecx,offset
                         mscorlib_ni+0x517edc (71b47edc) (MT: System.Random)
039b0498 e82b2cc1ff  call    035c30c8
                         (JitHelp: CORINFO_HELP_NEWSFAST)
039b049d 8bf0          mov     esi, eax
039b049f e8dc70326f  call    clr!SystemNative::GetTickCount
                         (72cd7580)
039b04a4 8bd0          mov     edx, eax
039b04a6 8bce          mov     ecx, esi
039b04a8 e817d40c6e  call    mscorlib_ni+0x44d8c4 (71a7d8c4)
                         (System.Random..ctor(Int32), mdToken: 060010dc)
...
```

Summary

To minimize the impact of JIT, carefully examine any areas of large amounts of generated code, whether from regular expressions, code generation, `dynamic`, or any other source. Use profile-guided optimization to decrease application startup time by pre-JITting the most useful code in parallel. Ensure you are using the latest version of .NET to take advantage of improvements in the JIT compiler.

To encourage function inlining, avoid things like virtual methods, loops, exception handling, recursion, or large method bodies. But do not sacrifice the integrity of your application by over-optimizing in this area.

Consider using NGEN for large applications or situations where you cannot afford the JIT cost during startup. Use MPGO to optimize the native images before using NGEN.

For Windows Store applications, ensure you are using .NET Native.

When nothing else works, develop a custom warmup strategy for your application to exercise your hot paths before they are needed for real work.

Chapter 4

Asynchronous Programming

With the ubiquity of multicore processors in today's computers, even on small devices such as cell phones, the ability to program effectively for multiple threads is a critical skill for all programmers.

There are essentially three reasons for using multiple threads:

1. You do not want to block the main thread of a UI with some background work.
2. You have so much work to do that you cannot afford to waste CPU time waiting for I/O to complete.
3. You want to use all of the processors at your disposal.

The first reason does not have as much to do with performance as it does with not annoying the end user. The 2nd and 3rd options are all about efficient use of computing resources.

Computer processors have effectively hit a wall in terms of raw clock speed. The main technique we are going to use in the foreseeable future to achieve higher computing throughput is parallelism. Taking advantage of multiple processors is critical to writing a high-performance application, particularly servers handling many simultaneous requests.

There are a few ways to execute code concurrently in .NET. For example, you can manually spin up a thread and assign it a method to run. This is fine for a relatively long-running method, but for many things, dealing with threads directly is very inefficient. If you want to execute a lot of short-running tasks, for example, the overhead of scheduling individual threads could easily outweigh the cost of actually running your code. To understand why, you need to know how threads are scheduled in Windows.

Each processor can execute only a single thread at a time. When it comes time to schedule a thread for a processor, Windows needs to do a context switch. During a context switch, the Windows kernel saves the processor's current thread's state to the operating system's internal thread object, picks from all the highest-priority ready threads, transfers the thread's context information from the thread object to the processor, and then finally starts it executing. If Windows switches to a thread from a different process, even more expense is incurred as the address space is swapped out.

The thread will then execute the code for the thread quantum, which is a multiple of the clock interval (the clock interval is about 15 milliseconds on current multiprocessor systems and cannot easily be changed, nor is it recommended to do so). When the code returns from the top of its stack, enters a wait state, or the time quantum is expired, the scheduler will pick another ready thread to execute. It may be the same thread, or not, depending on the contention for the processors. A thread can enter a wait state if it blocks on I/O of any kind, or voluntarily enters this state by calling `Thread.Sleep`.

Note

Windows Server has a higher thread quantum than the desktop version of Windows, meaning it runs threads for longer periods of times before a context switch. This setting can be controlled to some extent in the Performance Options of Advanced System Settings. Setting the option to Background Services will increase the default thread quantum for the system, possibly

at the expense of program responsiveness. These settings are stored in the registry, but you should not manipulate them directly.

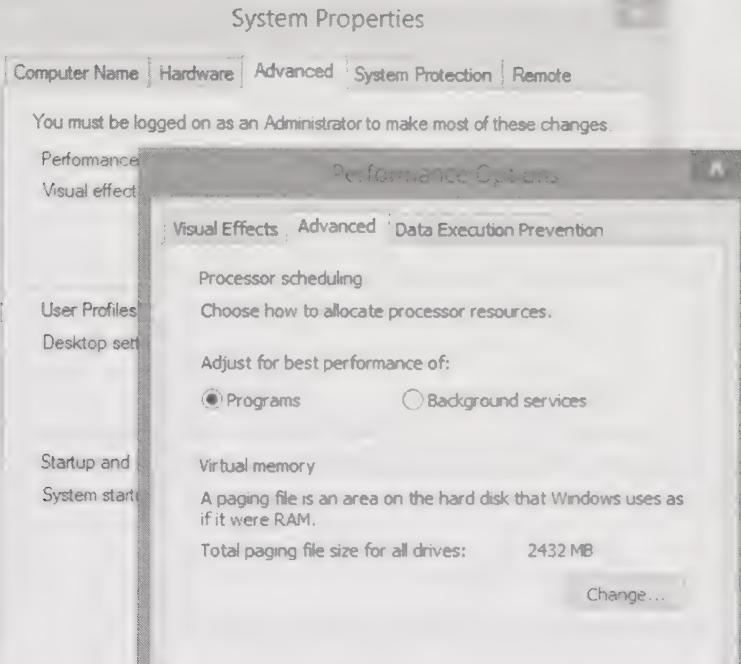


Figure 4.1. Decide whether a machine should prioritize foreground or background tasks.

The Thread Pool

Creating new threads is an expensive procedure. It requires allocation in the kernel, committing memory for stack space, more context switching, and possibly triggering per-thread initialization code in multiple DLLs. Because of this expense, you should never create a thread to handle each individual request, for example. You should treat threads as a pooled, shareable resource. Once available, they can be reused for the next piece of work. Thankfully, .NET makes

this easy and manages a thread pool for each managed process. This pool maintains a list of threads, keeping those alive while unused. To use this pool, you queue callbacks to it:

```
static void Main()
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(MyFunc),
                                  "my data");
}

private static void MyFunc(Object obj)
{
    var data = obj as string;
    // do work
}
```

These threads are created as needed and stick around to handle future needs, avoiding the recreation cost. The program saves on the overhead of creation and deletion and there is almost always a thread ready to handle the asynchronous tasks that come up.

Internally, there are two actual pools:

1. Worker thread pool: Handles CPU-bound tasks, assigning threads to different cores.
2. I/O thread pool: Handles asynchronous I/O completion from hardware, such as when storage devices or the network return data.

The thread pool has a difficult job. It needs to ensure that there is an available thread to handle any tasks that come its way, but it also needs to ensure that there are not too many threads running, which can be equally disastrous for performance. The pool attempts to optimize itself for throughput, which is the number of tasks completed per unit of time. This metric can balance having too few threads (not enough work getting done) or too many threads (too much resource contention).

In a world of no contention, the ideal case would be for each processor to execute one thread. However, most threading tasks are imperfect and can have blocking, either for I/O or some other resource. Therefore, it would make sense for the thread pool to schedule more threads per core to ensure that each core stays busy. However, if it gets too busy, then more threads just become more overhead for the pool to deal with and the processor can spend more time switching than getting anything done. (To be clear, this can happen regardless of the source of threads, but the pool is the most common source.) It is a delicate balance.

The main way the thread pool accomplishes this balance is via Hill Climbing, a standard type of algorithm that relies on self-feedback to tune itself over time. The pool schedules threads to execute and at intervals monitors its throughput (completed tasks per interval). It will continue to add threads until it sees the throughput drop, and then it will start removing threads. Because throughput can change based on the type of workload, this simple approach results in a very noisy, sporadic fluctuation in the number of threads. To deal with this noise, the thread pool treats the throughput as a continuous signal and applies a Fourier transform to it that lessens the impact of small fluctuations. Even when the the thread pool stops scheduling as many threads, it may keep some around in memory just to avoid reallocating them later (thus why it is a pool).

The thread pool should be your default choice for thread scheduling, but there are cases where it is not appropriate. For example, it is not as useful for threads that require specific priority levels or execute long-running tasks. If you have many tasks that take more than a few hundred milliseconds to execute, the thread pool will not be as effective. There may also be situations where you want a dedicated thread for a particular task, long-running or not. For these cases, you should create and manage your own thread.

The Task Parallel Library

If your program consists of pure CPU tasks significantly longer than the length of the thread quantum, then creating and using threads directly is acceptable, though unnecessary as we will see. However, if your code consists of many smaller

tasks that are unlikely to take this long, using threads directly is inefficient because the program will spend a significant time context switching rather than executing its actual code. While you can utilize the thread pool directly, as described in the previous section, it is no longer recommended to do so. Instead, you can use **Task** objects for both long and short actions.

.NET 4.0 introduced an abstraction of threads called a **Task** as part of the Task Parallel Library (TPL), which is the preferred way of achieving parallelism in most .NET code. The TPL gives you a lot of control over how your code is run, allows you to determine what happens when errors occur, provides the ability to conditionally chain multiple methods in sequence, and much more.

Internally, the TPL uses the .NET thread pool, but does so more efficiently, executing multiple **Task** delegates on the same thread sequentially before returning the thread back to the pool. It can do this via intelligent use of delegate objects. This effectively avoids the problem of wasting a thread's quantum with a single small task and causing too many context switches.

The TPL is a large and comprehensive set of APIs, but it is easy to get started. The basic principle is that you pass a delegate to the **Task**'s **Start** method. You also optionally call **ContinueWith** on the **Task** and pass a second delegate, which is executed once the **Task** is complete. Mastering the various execution and continuation options is crucial for extracting the most performance with minimal overhead. In this section I will briefly discuss the most common ways of manipulating tasks to set a baseline.

You can execute **Task** objects with both CPU-bound computations as well as I/O. All of these first examples will show pure CPU processing. There is a section dedicated to effective I/O later in this chapter.

The following code listing from the Tasks sample project demonstrates creating a **Task** for each processor. When each **Task** completes, it schedules for execution a continuation **Task** that has a reference to a callback method.

```
class Program
{
    static Stopwatch watch = new Stopwatch();
    static int pendingTasks;
```

```
static void Main(string[] args)
{
    const int MaxValue = 1000000000;

    watch.Restart();
    int numTasks = Environment.ProcessorCount;
    pendingTasks = numTasks;
    int perThreadCount = MaxValue / numTasks;
    int perThreadLeftover = MaxValue % numTasks;

    var tasks = new Task<long>[numTasks];

    for (int i = 0; i < numTasks; i++)
    {
        int start = i * perThreadCount;
        int end = (i + 1) * perThreadCount;
        if (i == numTasks - 1)
        {
            end += perThreadLeftover;
        }
        tasks[i] = Task<long>.Run(() =>
        {
            long threadSum = 0;
            for (int j = start; j <= end; j++)
            {
                threadSum += (long)Math.Sqrt(j);
            }
            return threadSum;
        });
        tasks[i].ContinueWith(OnTaskEnd);
    }
}

private static void OnTaskEnd(Task<long> task)
{
    Console.WriteLine("Thread sum: {0}", task.Result);
    if (Interlocked.Decrement(ref pendingTasks) == 0)
    {
        watch.Stop();
        Console.WriteLine("Tasks: {0}", watch.Elapsed);
    }
}
```

If your continuation Task is a fast, short piece of code, you should specify that it runs on the same thread as its owning Task. This is vitally important in an extremely multithreaded system as it can be a significant waste to spend time queuing Task objects to execute on a separate thread which may involve a context switch.

```
task.ContinueWith(OnTaskEnd,
    TaskContinuationOptions.ExecuteSynchronously);
```

If the continuation Task is called back on an I/O thread, then you might not want to use `TaskContinuationOptions.ExecuteSynchronously` as this could tie up an I/O thread that you need for pulling data off the network. As always, you will need to experiment and measure the result carefully. It is often more efficient for the I/O thread to just do the quick continuation work and avoid the extra scheduling.

If you do need a long-running Task, pass the `TaskCreationOptions.LongRunning` flag to the `Task.Factory.StartNew` method. There is also a version of this flag for continuations:

```
var task = Task.Factory.StartNew(action,
    TaskCreationOptions.LongRunning);
task.ContinueWith(OnTaskEnd, TaskContinuationOptions.LongRunning);
```

Continuations are the real power of the TPL. You can do all sorts of complex things that are outside the scope of performance and I will mention a few of them briefly here.

You can execute multiple continuations for a single Task:

```
Task task = ...
task.ContinueWith(OnTaskEnd);
task.ContinueWith(OnTaskEnd2);
```

`OnTaskEnd` and `OnTaskEnd2` have no relationship with each other and execute independently, and insofar as possible, in parallel.

On the other hand, you can also chain continuations:

```
Task task = ...  
task.ContinueWith(OnTaskEnd).ContinueWith(OnTaskEnd2);
```

Chained continuations have serial dependency relationships with each other. When a task ends, `OnTaskEnd` will run. Once that completes, `OnTaskEnd2` will execute.

Continuations can be told to execute only when their antecedent Task ends successfully (or fails, or is canceled, etc.):

```
Task task = ...  
task.ContinueWith(OnTaskEnd,  
                  TaskcontinuationOptions.OnlyOnRanToCompletion);  
task.ContinueWith(OnTaskEnd,  
                  TaskContinuationOptions.NotOnFaulted);
```

You can invoke a continuation only when multiple Task objects are completed (or any one of them has):

```
Task[] tasks = ...  
Task.Factory.ContinueWhenAll(tasks, OnAllTaskEnded);  
Task.Factory.ContinueWhenAny(tasks, OnAnyTaskEnded);
```

Using these APIs greatly simplifies how you can ensure that large swaths of your program remain 100% asynchronous without any blocking calls or unnecessary sync points.

Task Cancellation

To cancel a Task that is already running requires some cooperation. It is never a good idea to force-terminate a thread, and the Task Parallel Library does not allow you to access the underlying thread, let alone abort it. (If you do program directly with a `Thread` object, then you can call the `Abort` method, but this is dangerous and not recommended. Just pretend this API does not exist. Aborting a thread can leave synchronization objects in unknown states, shared state could be corrupted, and asynchronous operations may never complete.)

To cancel a Task, you need to pass a CancellationToken object to the delegate where it can then poll the token to determine if it needs to end processing. This example also demonstrates using a lambda expression as a Task delegate.

You can find this code in the sample TaskCancellation project.

```
static void Main(string[] args)
{
    var tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;

    Task task = Task.Run(() =>
    {
        while (true)
        {
            // do some work...
            if (token.IsCancellationRequested)
            {
                Console.WriteLine("Cancellation requested");
                return;
            }
            Thread.Sleep(100);
        }
    }, token);

    Console.WriteLine("Press any key to exit");

    Console.ReadKey();

    tokenSource.Cancel();

    task.Wait();

    Console.WriteLine("Task completed");
}
```

Handling exceptions

If a Task throws an exception during its execution, what happens depends on how the Task and its result are handled. The naive way is to just access the Result property. If there is no result because the Task threw an exception, then the thread that accesses the Result property or calls Wait explicitly will get an `AggregateException`.

```
var task = Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
});
task.Wait(); // exception here!
int result = task.Result;
```

If you are not explicitly waiting, but have a continuation, you have the same problem:

```
Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
}).ContinueWith(task =>
{
    int val = task.Result; //exception here!
});
```

There are a few solutions. First, you can wrap the Result access in an exception handler:

```
Task<int>.Factory.StartNew(() =>
{
    int x = 42;
    int y = 0;
    return x / y;
}).ContinueWith(task =>
```

```
{  
    try  
    {  
        // safely handle result  
        int val = task.Result;  
    }  
    catch(AggregateException ex)  
    {  
        LogException(ex);  
    }  
};
```

But we will learn later that throwing exceptions is bad. The Task class provides a number of properties to determine the state of a task:

- **IsCanceled**: The task was canceled.
- **IsFaulted**: An exception was thrown. The exception can be accessed via the **Exception** property.
- **IsCompleted**: True when the task is done executing, whether due to a fault, cancellation, or successful completion.
- **IsCompletedSuccessfully**: True when the task has completed without fault or cancellation.

You can use **IsFaulted** to detect whether there are unhandled exceptions:

```
Task<int>.Factory.StartNew(() =>  
{  
    int x = 42;  
    int y = 0;  
    return x / y;  
}).ContinueWith(task =>  
{  
    if (task.IsFaulted)  
    {  
        LogException(task.Exception);  
    }  
    else
```

```
{  
    // safely handle result  
    int val = task.Result;  
}  
});
```

Unobserved Task Exceptions

What happens if you do not handle an exception in a Task? Before .NET 4.5, if a Task's exception went unobserved, it would crash the process. When it goes unobserved, an object will be put in the finalization queue that knows about it. The next time the garbage collector runs finalizers, the exception will be thrown. This can make understanding the actual crash stack slightly trickier to discover because the exception at that point has a stack originating in the finalization thread. You will need to dig into the exceptions included in the AggregateException's InnerExceptions property to see the real exceptions and retrieve the stacks from them.

If you want to merely see a standard string dump of a tree of exceptions, you can just call the `ToString` method on the `AggregateException`. This will contain all nested exceptions as well.

If you need more fine-grained control of the exception logging, you can use a method similar to this, which correctly handles `AggregateException`:

```
public static class ExceptionUtils  
{  
    public static void LogException(System.Exception exception)  
    {  
        LogExceptionRecursive(exception, 0, null);  
    }  
  
    private static void LogExceptionRecursive(System.Exception ex,  
                                              int recursionLevel)  
    {  
        if (ex == null)  
        {  
            return;  
        }
```

```
}

if (recursionLevel >= 10)
{
    // Just to be safe, have a recursion cutoff
    return;
}

Console.WriteLine(
    $"Type: {ex.GetType()}, Message: {ex.Message}, " +
    $" Stack: {ex.StackTrace}, Level: {recursionLevel}");

var aggEx = ex as AggregateException;
if (aggEx != null && aggEx.InnerExceptions != null)
{
    foreach (var inner in aggEx.InnerExceptions)
    {
        LogExceptionRecursive(inner, recursionLevel + 1);
    }
}
else if (ex.InnerException != null)
{
    LogExceptionRecursive(ex.InnerException,
                          recursionLevel + 1);
}
}
```

What counts as observing the exception?

- Calling `Wait` on a Task
- Accessing the `Result` property
- Accessing the `Exception` property

Starting in .NET 4.5, the default behavior of unobserved task exceptions has changed. By default, it no longer causes an exception to be thrown from the finalizer thread. This decision was made because so much more of the .NET Framework was using Task Parallel Library, and they did not want a large amount of existing software to start crashing after a .NET update.

Despite this, it is generally undesirable to allow exceptions to be swallowed in this manner. It indicates severe problems—undefined state, which can lead to hard-to-detect bugs that occur later, or even data corruption—that your application is ignoring. Thankfully, you can restore the previous behavior with a simple configuration switch:

```
<configuration>
    <runtime>
        <ThrowUnobservedTaskExceptions enabled="true"/>
    </runtime>
</configuration>
```

You should include this setting in all new projects to ensure that your program avoids executing after unhandled exceptions. The next question is how to actually catch and log these exceptions. If you do nothing, the operating system will capture a minidump, from which you can examine the exceptions, but if you want more, you need to setup the unhandled exception handler.

```
static void Main()
{
    AppDomain.CurrentDomain.UnhandledException
        += OnUnhandledException;
}

private static void OnUnhandledException(
    object sender,
    UnhandledExceptionEventArgs e)
{
    var ex = e.ExceptionObject as Exception;
    if (ex != null)
    {
        ExceptionUtils.LogException(ex,
            Events.Log.UnhandledException
            true);
    }
    else if (e.ExceptionObject != null)
    {
        var type = e.ExceptionObject.GetType().ToString();
        Console.WriteLine(
            $"Non-exception object: {type} - {e.ExceptionObject}");
    }
}
```

```
    else
    {
        Console.WriteLine("Unknown object");
    }
}
```

Child Tasks

When tasks are created from within another task, they are, by default, independent from each other. This example creates a child task that executes arbitrary code. Both parent and child tasks write to the console, but they are completely unrelated to each other and either one can return and complete first.

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    });
}).ContinueWith(task => Console.WriteLine("Parent done"));
```

This is usually what you want as it maintains strong asynchrony that is a good habit in this type of processing, but it is possible to create a stronger relationship between parent and child tasks. One way is to have the parent wait on the child task (or on the `Result` property of the child task):

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    });

    // Explicit Wait!
    childTask.Wait();
}).ContinueWith(task => Console.WriteLine("Parent done"));
```

This is less than ideal—it introduces a blocking call into an otherwise asynchronous process, which is a violation of the cardinal rule of efficient asynchronous programs. An alternative is to pass the `TaskCreationOptions.AttachedToParent` flag, telling the parent that it cannot complete until all of its children do.

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("Inside child");
    }, TaskCreationOptions.AttachedToParent);

}).ContinueWith(task => Console.WriteLine("Parent done"));
```

While useful in circumstances where you want to enforce some coordination between tasks, this option does introduce new complexities that need to be dealt with.

First of all, it is possible for parent tasks to deny children the ability to attach themselves:

```
Task.Factory.StartNew(() =>
{
    Task childTask = Task.Factory.StartNew(() =>
    {
        // do work
    }, TaskCreationOptions.AttachedToParent);

}, TaskCreationOptions.DenyChildAttach);
```

In this case, the parent's preferences are obeyed and the child's request to attach is ignored. It behaves the same as the first example in this section. When you start tasks via `Task.Run` instead of the `Task.StartNew` method used here, `TaskCreationOptions.DenyChildAttach` is set by default.

You must also handle exceptions. An exception in an attached child Task is automatically passed onto the parent Task to be handled by the parent Task's continuation method (or whatever thread waits on the task's result). A de-

tached child `Task` that throws an exception must be handled in the same way as any other code inside the parent task (check its `Exception` property, or wrap the `Result` property check in an exception handler).

Cancellation also becomes a bit trickier. To simplify things, you should use the same `CancellationToken` object for all tasks in a parent-child relationship. When cancellation is signaled, what happens is determined by where and when the cancellation happens:

- Parent cancels before child `Task` start: Child never starts.
- Parent cancels after child `Task` starts: Child `Task` executes until it checks for cancellation.
- Attached child cancels: A `TaskCanceledException` is passed up to the parent's `Task` and will be included in its `AggregateException`.

TPL Dataflow

Many applications follow a data processing pipeline, where pieces of data flow through various stages of transformation or analysis. For purposes of efficiency, performance, and maximizing resource usage, it is ideal to make these stages asynchronous to be able to handle multiple pieces of data within the pipeline at once. You could certainly design such a system using the Task Parallel Library yourself, but this would involve a complex system of continuations, synchronization, and coordination. There is another option: use TPL Dataflow.

Note

TPL Dataflow is distributed via the `Microsoft.Tpl.Dataflow` NuGet package. It does not ship with the .NET Framework.

With TPL Dataflow, you construct processing blocks and hook them together into a pipeline or network. You can control the parallelism of each block, and each block can be executed asynchronously. Blocks pass messages to each other, which are just objects that you declare via type specifiers. A block can be linked to multiple sources or multiple targets. All processing can happen asynchronously.

It is important to note that using TPL Dataflow does not automatically make your program faster. Rather, it makes it easier to achieve good performance for this style of algorithm by taking advantage of TPL and doing much of the boilerplate programming for you, allowing you to focus on the important logic of your application.

Before we look at a simple example, examine some of the types of blocks you can create. There are a number of pre-built blocks that TPL Dataflow provides for you:

TPL Dataflow Block Type	Description
<code>BufferBlock<T></code>	FIFO queue of messages
<code>BroadcastBlock<T></code>	Sends the most recent message to all targets
<code>WriteOnceBlock<T></code>	Similar to <code>BroadcastBlock</code> , but can only set the value once
<code>ActionBlock<T></code>	Executes a delegate for the input, produces no output
<code>TransformBlock<TInput, TOutput></code>	Executes a delegate that can return a different type than it ingests
<code>TransformManyBlock<TInput, TOutput></code>	Similar to <code>TransformBlock</code> , but can output multiple outputs for every input
<code>BatchBlock<T></code>	Converts multiple inputs into a single array output

You can create your own block types by implementing the `ISourceBlock<TOutput>` and/or `ITargetBlock<TInput>` interfaces.

A TPL Dataflow Example

To see how TPL Dataflow blocks can work together, we will create a simple example of a text processing pipeline. This application processes a directory of text files and analyzes the frequency of all the words in each file. There are some post-processing steps that combine the analyses and weed out some common, uninteresting words.

Here is the creation of the pipeline, which returns a reference to the first block:

```
private static readonly HashSet<string> IgnoreWords =
    new HashSet<string>() { "a", "an", "the", "and", "of", "to" };
private static readonly Regex WordRegex =
    new Regex("[a-zA-Z]+", RegexOptions.Compiled);

private static ITargetBlock<string> CreateTextProcessingPipeline(
    string inputPath,
    out Task completionTask)
{
    int fileCount = Directory.GetFiles(inputPath, "*.txt").Length;

    var getFilenames = new TransformManyBlock<string, string>(
        path =>
    {
        return Directory.GetFiles(path, "*.txt");
    });

    var getFileContents = new TransformBlock<string, string>(
        async (filename) =>
    {
        Console.WriteLine("Begin: getFileContents");
        using (var streamReader = new StreamReader(filename))
        {
            return await streamReader.ReadToEndAsync();
        }
        Console.WriteLine("Begin: getFileContents");
    }, new ExecutionDataflowBlockOptions
    {
        MaxDegreeOfParallelism = Environment.ProcessorCount
    });
}
```

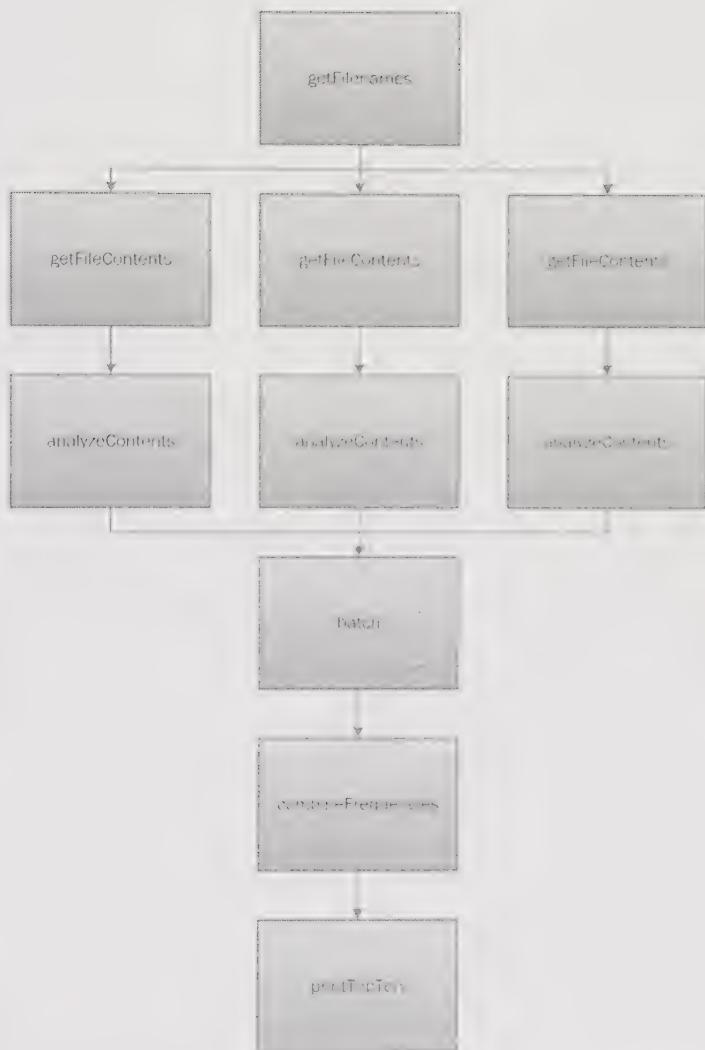


Figure 4.2. A graphical representation of the pipeline.

```
var analyzeContents =
    new TransformBlock<string, Dictionary<string, ulong>>(
contents =>
{
    Console.WriteLine("Begin: analyzeContents");
    var frequencies =
        new Dictionary<string, ulong>(
            10000, StringComparer.OrdinalIgnoreCase);

    var matches = WordRegex.Matches(contents);
    foreach (Match match in matches)
    {
        ulong currentValue;
        if (!frequencies.TryGetValue(match.Value,
                                      out currentValue))
        {
            currentValue = 0;
        }
        frequencies[match.Value] = currentValue + 1;
    }
    Console.WriteLine("End: analyzeContents");
    return frequencies;
}, new ExecutionDataflowBlockOptions
{
    MaxDegreeOfParallelism = Environment.ProcessorCount
});
};

var eliminateIgnoredWords =
    new TransformBlock<Dictionary<string, ulong>,
                      Dictionary<string, ulong>>(
input =>
{
    foreach(var word in IgnoreWords)
    {
        input.Remove(word);
    }
    return input;
});

var batch =
    new BatchBlock<Dictionary<string, ulong>>(fileCount);
```

```

// This is one point of synchronization --
// all processing converges on this
var combineFrequencies =
    new TransformBlock<Dictionary<string, ulong>[], 
                           List<KeyValuePair<string, ulong>>>(
inputs =>
{
    Console.WriteLine("Begin: combiningFrequencies");
    var sortedList = new List<KeyValuePair<string, ulong>>();
    var combinedFrequencies = new Dictionary<string, ulong>(
        10000, StringComparer.OrdinalIgnoreCase);

    foreach (var input in inputs)
    {
        foreach (var kvp in input)
        {
            ulong currentFrequency;
            if (!combinedFrequencies.TryGetValue(
                kvp.Key,
                out currentFrequency))
            {
                currentFrequency = 0;
            }
            combinedFrequencies[kvp.Key] = currentFrequency +
                kvp.Value;
        }
    }
    foreach(var kvp in combinedFrequencies)
    {
        sortedList.Add(kvp);
    }
    sortedList.Sort((a, b) =>
    {
        return -a.Value.CompareTo(b.Value);
    });
    Console.WriteLine("End: combineFrequencies");

    return sortedList;
}, new ExecutionDataflowBlockOptions()
{
    MaxDegreeOfParallelism = 1
});

```

```

var printTopTen = new ActionBlock<List<KeyValuePair<string,
                                             ulong>>>(
    input =>
{
    for (int i=0;i<10;i++)
    {
        Console.WriteLine(
            $"{input[i].Key} - {input[i].Value}");
    }
    getFilenames.Complete();
});

// Hook up blocks
getFilenames.LinkTo(getFileContents);
getFileContents.LinkTo(analyzeContents);
analyzeContents.LinkTo(eliminateIgnoredWords);
eliminateIgnoredWords.LinkTo(batch);
batch.LinkTo(combineFrequencies);
combineFrequencies.LinkTo(printTopTen);

completionTask = getFilenames.Completion;

return getFilenames;
}

```

To use the pipeline, you just need to retrieve the block reference and post a message to it:

```

Task completionTask;
ITargetBlock<string> startBlock =
    CreateTextProcessingPipeline(args[0], out completionTask);

startBlock.Post(args[0]);

completionTask.Wait();

```

In the output, you can see from the console output that there are some blocks which execute concurrently, while others execute serially after previous blocks complete, and different types of blocks can be interleaved:

```
Begin: getFileContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: analyzeContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: analyzeContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: analyzeContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: getFileContents  
Begin: analyzeContents  
Begin: analyzeContents  
End: analyzeContents  
Begin: analyzeContents  
End: analyzeContents  
Begin: analyzeContents  
End: analyzeContents  
Begin: analyzeContents  
End: analyzeContents
```

```
Begin: analyzeContents
End: analyzeContents
End: analyzeContents
Begin: combiningFrequencies
End: combineFrequencies
I - 7693
you - 5188
in - 4266
My - 4237
that - 4158
is - 3790
NOT - 3209
FOR - 3201
d - 3117
IT - 2947
```

Parallel Loops

One of the examples in the TPL section demonstrates a pattern that is so common that there is an API just for parallel execution of loops.

```
const int MaxValue = 1000;  
long sum = 0;
```

```
Parallel.For(0, MaxValue, (i) =>
{
    Interlocked.Add(ref sum, (long)Math.Sqrt(i));
});
```

There is also a version of `foreach` for handling generic `IEnumerable<T>` collections.

```
var urls = new List<string>
{
    @"http://www.microsoft.com",
    @"http://www.bing.com",
    @"http://msdn.microsoft.com"
};
var results = new ConcurrentDictionary<string, string>();
var client = new System.Net.WebClient();

Parallel.ForEach(urls, url => results[url] =
    client.DownloadString(url));
```

If you want to interrupt loop processing, you can pass a `ParallelLoopState` object to the loop delegate. There are two options for stopping the loop:

- **Break:** Tells the loop not to execute any iterations that are sequentially greater than the current iteration. In a `Parallel.For` loop, if you call `ParallelLoopState.Break` on the i^{th} iteration then any iterations less than i will still be allowed to run, but any iterations greater than i will be prevented from running. This also works on a `Parallel.ForEach` loop, but each item is assigned an index, and this may be arbitrary from the program's point of view. Note that it is possible for multiple loop iterations to call `Break`, depending on the logic you code inside the loop.
- **Stop:** Tells the loop not to execute any more iterations at all.

The following example uses `Break` to stop the loop at an arbitrary location.

```
Parallel.ForEach(urls, (url, loopState) =>
{
```

```
if (url.Contains("bing"))
{
    loopState.Break();
}
results[url] = client.DownloadString(url);
});
```

When using parallel loops, you want to ensure that each loop iteration operates on local state as much as possible. It is very easy to lose all of the benefits of parallelism if your loop spends all of its time blocked on synchronized access to a shared variable that all the threads use. If you must have shared state, then make sure that the amount of work you do per iteration is significantly larger than the amount of time you spend synchronizing that shared state.

Another problem with parallel loops is that a delegate is invoked for each iteration, which may be wasteful if the work to be done is less than the cost of a delegate or method invocation (See Chapter 5).

Both problems can be solved with the `Partitioner` class which transforms a range into a set of `Tuple` objects that each describe a sub-range to be iterated over on the original collection.

The following example demonstrates just how much synchronization can negatively affect the effectiveness of parallelism.

```
static void Main(string[] args)
{
    Stopwatch watch = new Stopwatch();
    const int.MaxValue = 1000000000;

    long sum = 0;

    // Naive For-loop
    watch.Restart();
    sum = 0;
    Parallel.For(0, MaxValue, (i) =>
    {
        Interlocked.Add(ref sum, (long)Math.Sqrt(i));
    });
    watch.Stop();
    Console.WriteLine("Parallel.For: {0}", watch.Elapsed);
```

```
// Partitioned For-loop
var partitioner = Partitioner.Create(0, MaxValue);
watch.Restart();
sum = 0;
Parallel.ForEach(partitioner,
    (range) =>
{
    long partialSum = 0;
    for (int i = range.Item1; i < range.Item2; i++)
    {
        partialSum += (long)Math.Sqrt(i);
    }
    Interlocked.Add(ref sum, partialSum);
});
watch.Stop();
Console.WriteLine("Partitioned Parallel.For: {0}",
    watch.Elapsed);
}
```

You can find this code in the ParallelLoops sample project. On one run on my machine, the output was this:

```
Parallel.For: 00:01:47.5650016
Partitioned Parallel.For: 00:00:00.8942916
```

The above partitioning scheme is static in that once the partitions are determined, a delegate executes each range and if one finishes early there is no attempt to repartition to get other processors working. You can create static partitions on any `IEnumerable<T>` collection without specifying a range, but there will be a delegate call for each item, not for a sub-range. It is possible to get around this by creating a custom `Partitioner`, which can be quite involved. For more information and to see some extensive examples, search for the article, “Custom Parallel Partitioning With .NET 4” by Stephen Toub .

Performance Tips

Avoid Blocking

To obtain the highest performance you must ensure that your program never wastes one resource while waiting for another. Most commonly, this takes the form of blocking the current thread while waiting for some I/O to complete. This situation will cause one of two things to happen:

1. The thread will be blocked in a waiting state, get unscheduled, and cause another thread to run. This could mean creating a new thread to handle pending work items or tasks if all current threads are in use or blocked.
2. The thread will hit a synchronization object which might spin for a few milliseconds waiting for a signal. If it does not get it in time, it will enter the same state as 1.

In both cases, it needlessly increases the size of the thread pool, and possibly also wastes CPU spinning in locks. Neither situation is desirable.

Locking and other types of direct thread synchronization are all explicit blocking calls and easy to detect. However, it is not always so obvious what other method calls may lead to blocking. They often revolve around I/O of various kinds so you need to make sure that any interactions with the network, the file system, databases, or any other high-latency service is done asynchronously. Thankfully, .NET makes it fairly easy to do that with `Task` objects.

When making use of any I/O API, whether it is for network, file system, databases, or anything else, make sure that it returns a `Task`; otherwise, it is highly suspect and probably doing blocking I/O. Note that older asynchronous APIs will return an `IAsyncResult` and usually start with `Begin-`. Either find an alternate API that returns a `Task` instead, or use the `Task.FactoryFromAsync` method to wrap these methods inside of `Task` objects to keep your own programming interface consistent.

You should rarely or never call the `Task.Wait` method. Use continuations instead. Waiting on a `Task` is a subset of the larger problem of blocking calls.

Avoid Lock and Dispatch Convoys

A common restriction for UI applications is that the UI state can only be modified from a single thread, often termed the “dispatcher” thread. This means, that even if you want multiple threads to do some background work, they must transition their work back to the UI thread to communicate any visible changes.

In WPF, it looks something like this:

```
// On a background worker thread
Application.Current.Dispatcher.BeginInvoke(
    DispatcherPriority.Background,
    () => this.statusBar.Value = "Complete");
```

If the UI application has many threads doing this kind of thing constantly, you can start to lock up the UI with multiple updates. In these kinds of cases, you will need to either rethink the UI changes themselves or introduce an aggregation layer where you can batch up multiple updates and update all parts of the UI at once.

This is similar to the lock convoy problem, where contention is so high for a single lock that more time is spent spinning or waiting for the lock to free than actually doing work. In both cases, you need to rethink the async strategy to reduce contention on the single resource.

Use Tasks for Non-Blocking I/O

.NET 4.5 added `-Async` methods to the `Stream` class so that now all `Stream`-based communication can be entirely asynchronous quite easily. Here is a simple example:

```
int chunkSize = 4096;
var buffer = new byte[chunkSize];

var fileStream = new FileStream(filename, FileMode.Open,
    FileAccess.Read, FileShare.Read, chunkSize, useAsync: true);
```

```
var task = fileStream.ReadAsync(buffer, 0, buffer.Length);
task.ContinueWith((readTask) =>
{
    int amountRead = readTask.Result;
    fileStream.Dispose();
    Console.WriteLine("Async(Simple) read {0} bytes", amountRead);
});
```

You can no longer take advantage of the using syntax to clean up IDisposable objects such as Stream objects. Instead, you must pass those objects to the continuation method to make sure they are disposed along every path.

The above example is actually quite incomplete. In a real scenario, you will often have to make multiple reads to a stream to get the full contents. This can happen if the files are larger than the buffer you provide, or if you are dealing with a network stream instead of files. In that case, the bytes have not even arrived at your machine yet. To handle this situation asynchronously, you need to continue reading the stream until it tells you there is no data left.

An additional wrinkle is that now you need two levels of Task objects. The top level is for the overall read—the portion your calling program is interested in. The level below that is the set of Task objects for each individual chunked read.

Consider why this is so. The first asynchronous read will return a Task. If you return that up to the caller to wait or continue on, then they will continue executing after the first read is done. What you really want is for them to continue executing after all the reads are complete. This means you cannot return that first Task back to the caller. You need a fake Task that completes once all the reads are done.

To accomplish all of this, you will need to use the TaskCompletionSource<T> class, which can generate that fake Task for you to return. When your series of asynchronous reads are complete, you call the TrySetResult method on the TaskCompletionSource, which will cause it to trigger whoever is waiting or continuing on it.

The following example expands on the previous example and demonstrates the use of TaskCompletionSource:

```
private static Task<int> AsynchronousRead(string filename)
{
    int chunkSize = 4096;
    var buffer = new byte[chunkSize];
    var tcs = new TaskCompletionSource<int>();

    var fileContents = new MemoryStream();
    var fileStream = new FileStream(filename, FileMode.Open,
        FileAccess.Read, FileShare.Read, chunkSize, useAsync: true);
    fileContents.Capacity += chunkSize;

    var task = fileStream.ReadAsync(buffer, 0, buffer.Length);
    task.ContinueWith(
        readTask =>
        ContinueRead(readTask, fileStream,
            fileContents, buffer, tcs));

    return tcs.Task;
}

private static void ContinueRead(Task<int> task,
    FileStream stream,
    MemoryStream fileContents,
    byte[] buffer,
    TaskCompletionSource<int> tcs)
{
    if (task.IsCompleted)
    {
        int bytesRead = task.Result;
        fileContents.Write(buffer, 0, bytesRead);
        if (bytesRead > 0)
        {
            // More bytes to read, so make another async call
            var newTask = stream.ReadAsync(buffer, 0, buffer.Length);
            newTask.ContinueWith(
                readTask =>
                ContinueRead(readTask, stream,
                    fileContents, buffer, tcs));
        }
        else
        {
            // All done, dispose of resources and
        }
    }
}
```

```
// complete top-level task.  
tcs.TrySetResult((int)fileContents.Length);  
stream.Dispose();  
fileContents.Dispose();  
}  
}  
}
```

Note that there is no requirement for anyone to wait on a Task before the result is set – no ordering dependency is a fairly important requirement for asynchronous coding.

Adapt the Asynchronous Programming Model to Tasks

Older style asynchronous methods in the .NET Framework have methods prefixed with `Begin-` and `End-`. These methods continue to work fine and can be easily wrapped inside a `Task` for a consistent interface, as in the following example, taken from the `TaskFromAsync` sample project:

```
const int TotalLength = 1024;  
const int ReadSize = TotalLength / 4;  
  
static Task<string> GetStringFromFileBetter(string path)  
{  
    var buffer = new byte[TotalLength];  
  
    var stream = new FileStream(  
        path,  
        FileMode.Open,  
        FileAccess.Read,  
        FileShare.None,  
        buffer.Length,  
        FileOptions.DeleteOnClose | FileOptions.Asynchronous);  
  
    var task = Task<int>.Factory.FromAsync(  
        stream.BeginRead,  
        stream.EndRead,  
        buffer,  
        0,  
        ReadSize, null);
```

```
var tcs = new TaskCompletionSource<string>();

task.ContinueWith(readTask => OnReadBuffer(readTask,
                                              stream, buffer, 0, tcs));

return tcs.Task;
}

static void OnReadBuffer(Task<int> readTask,
                        Stream stream,
                        byte[] buffer,
                        int offset,
                        TaskCompletionSource<string> tcs)
{
    int bytesRead = readTask.Result;
    if (bytesRead > 0)
    {
        var task = Task<int>.Factory.FromAsync(
            stream.BeginRead,
            stream.EndRead,
            buffer,
            offset + bytesRead,
            Math.Min(buffer.Length - (offset + bytesRead), ReadSize),
            null);

        task.ContinueWith(
            callbackTask => OnReadBuffer(
                callbackTask,
                stream,
                buffer,
                offset + bytesRead,
                tcs));
    }
    else
    {
        tcs.TrySetResult(Encoding.UTF8.GetString(buffer, 0, offset));
    }
}
```

The `FromAsync` method takes as arguments the stream's `BeginRead` and `EndRead` methods as well as the target buffer to store the data. It will execute the methods and, after `EndRead` is done, call the continuation, passing control back to your code, which in this example closes the stream and returns the converted file contents.

Use Efficient I/O

Just because you are using asynchronous programming with all I/O calls does not mean you are making the most out of the I/O you are doing. I/O devices have different capabilities, speeds, and features which means that you often need to tailor your programming to them.

In the examples above, I chose a 16KB buffer size for reading and writing to the disk. Is this a good value? Considering the size of buffers on hard disks and the speed of solid state devices, maybe not. Experimentation is required to figure out how to chunk the I/O efficiently. The smaller your buffers, the more overhead you will have. The larger your buffers, the longer you may need to wait for results to start coming in. The rules that apply to disks will not apply to network devices and vice-versa.

The most crucial thing, however, is that you also need to structure your program to take advantage of I/O. If part of your program ever blocks waiting for I/O to finish then that is time not spent crunching useful data with the CPU, or at least it is wasting the thread pool. While waiting for I/O to complete, do as much other work as possible.

Also, note that there is a huge difference between true asynchronous I/O and performing synchronous I/O on another thread. In the former case, you have actually handed control to the operating system and hardware and no code anywhere in the system is blocked waiting for it to come back. If you do synchronous I/O on another thread, you are just blocking a thread that could be doing other work while still waiting for the operating system to get back to you. This might be acceptable in a non-performance situation (e.g., doing background I/O in a UI program instead of the main thread), but it is never recommended.

In other words, the following example is bad and defeats the purpose of asynchronous I/O:

```
Task.Run( ()=>
{
    using (var inputStream = File.OpenRead(filename))
    {
        byte[] buffer = new byte[16384];
        // Calling a synchronous I/O method
        // -- this will block the thread.
        var input = inputStream.Read(buffer, 0, buffer.Length);
        ...
    }
});
```

For additional tips on doing effective I/O with the .NET Framework APIs specifically for disk or network access, see Chapter 6.

async and await

In .NET 4.5, there are two new keywords that can simplify your code in many situations: `async` and `await`. Used together, they turn your TPL code into something that looks like easy, linear, synchronous code. Under the hood, however, it is really using `Task` objects and continuations.

The following example comes from the `AsyncAwait` sample project.

```
static Regex regex = new Regex("<title>(.*)</title>",
                             RegexOptions.Compiled);

private static async Task<string> GetWebpageTitle(string url)
{
    System.Net.Http.HttpClient client =
        new System.Net.Http.HttpClient();
    Task<string> task = client.GetStringAsync(url);

    // now we need the result so await
    string contents = await task;
    Match match = regex.Match(contents);
```

```

if (match.Success)
{
    return match.Groups[1].Captures[0].Value;
}
return string.Empty;
}

```

To see where the real power of this syntax lies, consider a more complex example, where you are simultaneously reading from one file and writing to another, compressing the output along the way. It is not very difficult to use Task objects directly to accomplish this, but consider how trivial it looks when you use the `async/await` syntax. The following is from the CompressFiles sample project.

First, the synchronous version for comparison:

```

private static void SyncCompress(IEnumerable<string> fileList)
{
    byte[] buffer = new byte[16384];
    foreach (var file in fileList)
    {
        using (var inputStream = File.OpenRead(file))
        using (var outputStream = File.OpenWrite(file + ".compressed"))
        using (var compressStream = new GZipStream(outputStream,
                                                     CompressionMode.Compress))
        {
            int read = 0;
            while ((read = inputStream.Read(buffer,
                                              0,
                                              buffer.Length)) > 0)
            {
                compressStream.Write(buffer, 0, read);
            }
        }
    }
}

```

To make this asynchronous, all we have to do is add the `async` and `await` keywords and change the `Read` and `Write` methods to `ReadAsync` and `WriteAsync`, respectively:

```

private static async Task AsyncCompress(
    IEnumerable<string> fileList)
{
    byte[] buffer = new byte[16384];
    foreach (var file in fileList)
    {
        using (var inputStream = File.OpenRead(file))
        using (var outputStream = File.OpenWrite(file + ".compressed"))
        using (var compressStream =
            new GZipStream(outputStream,
                            CompressionMode.Compress))
        {
            int read = 0;
            while ((read = await inputStream.ReadAsync(buffer, 0,
                                              buffer.Length)) > 0)
            {
                await compressStream.WriteAsync(buffer, 0, read);
            }
        }
    }
}

```

It looks like this code will get blocked waiting for the HTTP result, but do not confuse `await` for “wait”; they are similar, but deliberately just different enough. Everything before the `await` keyword happens in the calling thread. Everything from the `await` onwards is in the continuation. Your code can `await` any method that returns a `Task<T>` as long as it is in a method marked `async`. With these keywords, the compiler will do the heavy lifting of transforming your code into a structure similar to the previous TPL examples.

Significantly, notice that there is an `await` within a `using` statement. The dispose pattern works correctly with `async` and `await`, greatly simplifying your code.

Using `async/await` can dramatically simplify your code, but there are some Task-based situations for which they cannot be used. For example, if the completion of a `Task` is nondeterministic, or you must have multiple levels of `Task` objects and are using `TaskCompletionSource<T>`, then `async/await` may not fit.

I ran into this determinism problem myself when I was implementing retry functionality on top of .NET's HTTP client functionality, which is fully `Task`-enabled. I started with a simple HTTP client wrapper class and I used `async/await` initially because it simplified the code. However, when it came time to actually implement the retry functionality, I immediately knew I was stuck because I had given up control of when the tasks completed. For my implementation, I wanted to send the retry request before the first request had actually timed out. Whichever request finished first is the one I wanted to return up to the caller. Unfortunately, `async/await` will not handle this nondeterministic situation of arbitrarily choosing from multiple child `Task` objects that you are waiting on. One way to resolve this is to use the `ContinueOnAny` method described earlier. Alternatively, you could use `TaskCompletionSource` to manually control when the top-level `Task` is complete. I chose the latter for my situation.

A Note On Program Structure

There is a crucial tidbit from the previous section: all `await` statements must be in methods marked `async`, which means those methods must return `Task` objects. This kind of restriction does not technically exist if you are using `Task` objects directly, but the same idea applies. Using asynchronous programming is like a beneficial “virus” that infects your program at all layers. Once it takes root in one part, it will move up the layers of function calls as high as it can.

Of course, using `Task` objects directly, you can create this (bad) example:

```
Task<string> task = Task<string>.Run(()=> { ... });
task.Wait();
```

But unless you are doing only trivial multithreaded work, this completely ruins the scalability of your application. Yes, you could insert some work between the task's creation and the call to `Wait`, but this is missing the point. You should almost never wait on `Task` objects. That wastes a thread that could otherwise be doing useful work. It may lead to more context switching and higher overhead from the thread pool as more threads are needed to handle the available work items.

If you carry this idea of never waiting to its logical conclusion, you will realize that it is very possible (even likely) that nearly all of your program's code will occur in a continuation of some sort. This makes intuitive sense if you think about it. A UI program does essentially nothing until a user clicks or types—it is reacting to input via an event mechanism. A server program is analogous, but instead of mouse and keyboard, the I/O is via the network or file system.

As a result, a high-performance application can easily start to feel disjointed as you split the program logic based on I/O boundaries. The earlier you plan for this, the better off you will be. It is critical to settle on just a few standard patterns that most or all of your program uses. For example:

- Determine where `Task` and continuation methods live. Do you use a separate method or a lambda expression? Does it depend on its size?
- If you use methods for continuations, settle on a standard prefix (e.g., `OnMyTaskEnd`).
- Standardize error handling. Do you have one continuation that handles all errors, cancellations, and normal completions? Or do you have separate methods to handle each of these and use `TaskContinuationOptions` to selectively execute them?
- Decide whether to use `async/await` or `Task` objects directly.
- If you have to call old style `Begin.../End...` asynchronous methods, wrap them in `Task` objects to standardize your handling, as described earlier.
- Do not feel like you have to use every feature of the TPL. Some features are not recommended for most situations (`AttachedToParent` tasks, for example). Standardize on the minimal feature set you can get away with.

Asynchronous programming really is a different kind of programming than standard, synchronous, straight-line procedures and calls. It requires you to develop a new mindset, one that is aggressive about eliminating waits and blocking calls and considers program structure in terms of independent pieces that only occasionally come together to synchronize. It can take a while to develop this kind of mindset. Start small and build up.

Use Timers Correctly

To schedule a method to execute after a certain timespan and optionally at regular intervals thereafter, use the `System.Threading.Timer` class. You should not use mechanisms like `Thread.Sleep` that block the thread for the specified amount of time, though we will see below that it can be useful in some situations.

The following example demonstrates how to use a timer by specifying a callback method and passing it two timeout values. The first value is the time until the timer fires for the first time and the second value is how often to repeat it thereafter. You can specify `Timeout.Infinite` (which has value -1) for either value. This example fires the timer only once, after 15 milliseconds.

```
private System.Threading.Timer timer;

public void Start()
{
    this.timer = new Timer(TimerCallback,
                          null,
                          15,
                          Timeout.Infinite);
}

private void TimerCallback(object state)
{
    // Do your work
}
```

Do not create an excessive number of timers. All `Timer` objects are serviced from a single thread in the thread pool. A huge number of `Timer` instances will cause delays in executing their callbacks. When the time comes due, the timer thread will schedule a work item to the thread pool, and it will be picked up by the next available thread. If you have a large number of tasks, queued work items, or high CPU usage, this means your `Timer` callbacks will not be accurate. In fact, they are guaranteed to never be more accurate than the operating system's timer tick count, which is set to 15.625 milliseconds by default. This is the same value that determines thread quantum lengths. Setting a timeout value less than that will not get you the results you need. If you need more precision than 15 milliseconds, you have a few options:

1. Reduce the operating system's timer tick resolution. This can result in higher CPU usage and severely impact battery life, but may be justified in some situations. Note that changing this could have far-reaching impacts like more context switches, higher system overhead, and worse performance in other areas.
2. Spin in a loop, using a high-resolution timer (see Chapter 6) to measure elapsed time. This also uses more CPU and power, but is more localized.
3. Call `Thread.Sleep`. This will suspend the thread and there is no guarantee that it will be resumed at the time you desire. On a highly loaded system, it is possible the thread could be context-switched out and you will not get it back until well after your desired interval.

When you use a `Timer`, be aware of a classic race condition. Consider the code:

```
private System.Threading.Timer timer;

public void Start()
{
    this.timer = new Timer(TimerCallback,
                          null,
                          15,
                          Timeout.Infinite);
}
```

```
private void TimerCallback(object state)
{
    // Do your work
    this.timer.Dispose();
}
```

This code sets up a `Timer` to execute a callback in 15 milliseconds. The callback just disposes the `timer` object once it is done with it. This code is also likely to throw a `NullReferenceException` in `TimerCallback`. The reason is that it is very possible for the callback to execute before the `Timer` object is assigned to the `this.timer` field in the `Start` method. Thankfully, it is quite easy to fix:

```
this.timer = new Timer(TimerCallback,
                      null,
                      Timeout.Infinite,
                      Timeout.Infinite);
this.timer.Change(15, Timeout.Infinite);
```

Note

A project I once worked on allocated a `System.Threading.Timer` object to track timeouts for each node in dependency graph. This was fine while the graphs were small. Once they started hitting a few thousands nodes in size, with hundreds of concurrently running nodes, we saw significant lock contention and unreliable timeout behavior. It quickly became an untenable, very expensive design. Instead, we designed our own timeout monitoring system that relied on polling at regular intervals—essentially, we reduced our design to use a single timer that checked all running nodes at a suitable frequency.

If you only need to run the target method a single time (i.e., no recurring scenarios), then you can use `Task.Delay` to schedule a delegate to run once in the future:

```
var task = Task.Delay(1000).ContinueWith(_ =>
{
    Console.WriteLine("After delay");
});
```

Ensure Good Thread Pool Size

The thread pool tunes itself over time, but at startup it has no history and will start in a default state. If your software is extremely asynchronous and uses a lot of CPU, then it may be hit with a steep startup cost as it waits for more threads to be created and become available. To reach the steady state sooner, you can tweak the startup parameters to maintain a minimum number of ready threads upon startup.

```
const int MinWorkerThreads = 25;
const int MinIoThreads = 25;
ThreadPool.SetMinThreads(MinWorkerThreads, MinIoThreads);
```

You do need to exercise caution here. If you are using `Task` objects, then they will be scheduled based on the number of threads available for scheduling. If there are too many threads then the `Task` objects can become over-scheduled, leading to less CPU efficiency, not more, as more context switching happens. The thread pool will still eventually switch to using an algorithm that can reduce the number of threads to below this number, if the work load is lighter.

You can also set a maximum with the `SetMaxThreads` method and this has the same risks.

To figure out how many threads you really need, leave this setting alone and analyze your app during its steady state using the `ThreadPool.GetMaxThreads` and `ThreadPool.GetMinThreads` methods, or by using performance counters to examine how many threads are in the process.

Do Not Abort Threads

Terminating threads without their cooperation is a dangerous procedure. Threads have their own clean up to do and calling `Abort` on them does not allow a graceful shutdown. When you kill a thread, you leave portions of the application in an undefined state. It would be better to just crash the program, but ideally, just do a clean restart.

To safely end a thread, you must use some shared state and the thread function itself must check that state to determine when it should end. It must be cooperative to be safe.

If you are using `Task` objects, as you should, there is no API provided to terminate a `Task`. Use a `CancellationToken`, as discussed earlier in this chapter, to allow cooperative termination.

Do Not Change Thread Priorities

In general, it is a bad idea to change thread priorities. Windows schedules threads according to their priority. If high-priority threads are always ready to run, then lower priority threads will be starved and rarely given a chance to run. By increasing a thread's priority you are saying that its work must have priority over all other work, including other processes. This notion is not a safe part of a stable system.

It is more acceptable to lower a thread's priority if it is doing something that can wait until all the normal-priority tasks have completed. One good reason to lower a thread's priority is that you have detected a runaway thread doing an infinite loop. You cannot safely terminate threads, so the only way to get that thread and processor resource back is by restarting the process. Until you can shut down and do that cleanly, lowering the runaway thread's priority is a reasonable mitigation. Note that even threads with lower priorities are still guaranteed to run after a while because Windows will increase their dynamic priority the longer it goes without executing. The exception is Idle priority (`THREAD_PRIORITY_IDLE`), which the operating system will only ever schedule if literally nothing else can run.

You may find a legitimate reason to increase thread priority, such as reacting to rare situations that require faster responses, but this should be used with care. Windows schedules threads agnostic of the process they belong to, so a high-priority thread from your process will run at the expense not only of your other threads, but all the other threads from other applications running on your system.

If you are using the thread pool, then any thread priority changes are reset every time the thread reenters the pool. If you are using the Task Parallel Library and still manipulating the underlying thread, then keep in mind that the same thread can run multiple tasks before being placed back in the pool.

Thread Synchronization and Locks

As soon as you start talking about multiple threads, the question of synchronization among those threads is going to appear. Synchronization is the practice of ensuring that only a single thread can access some shared state, such as a class's field. Thread synchronization is usually accomplished with synchronization objects such as `Monitor`, `Semaphore`, `ManualResetEvent`, and others. In aggregate, these are sometimes informally referred to as locks, and the process of synchronizing in a specific thread as locking.

One of the fundamental truths of locks is: Locking never improves performance. At best, it can be neutral, with a well-implemented synchronization primitive and no contention. A lock explicitly stops other threads from doing productive work, wasting CPU, increasing context switching time, and more. We tolerate this because the one thing more critical than raw performance is correctness. It does not matter how fast you calculate a wrong result!

Before attacking the problem of which locking apparatus to use, first consider some more fundamental principles.

Do I Need to Care About Performance At All?

Validate that you actually need to care about performance in the first place. This goes back to the principles discussed in the first chapter. Not all code is equal in your application. Not all of it should be optimized to the n^{th} degree. Typically, you start at the “inner loop” – the code that is most commonly executed or is most performance critical – and work outwards until the cost outweighs the benefit. There are many other areas in your code that are much less critical, performance-wise. In these situations, if you need a lock, take a lock and do not worry about it.

Now, you do have to be careful here. If your non-critical piece of code is executing on a thread pool thread, and you block it for a significant amount of time, the thread pool could start injecting more threads to keep up with other requests. If a couple of threads do this once in a while, it is no big deal. However, if you have lots of threads doing things like this, then it could become a problem because it wastes resources that should be doing real work. If you are running a program that processes a heavy, constant workload, then even parts of your program that are not performance-sensitive can negatively affect the system by context switching or perturbing the thread pool if you are not careful. As in all things, you must measure.

Do I Need a Lock At All?

The most performant locking mechanism is one that is not there. If you can completely remove your need for thread synchronization, then you are in the best place as far as performance goes. This is the ideal, but it might not be easy. It usually means you have to ensure there is no mutable shared state – each request through your application can be handled without regard to another request or any centralized volatile (read/write) data. If you can do this, this is the optimal performance scenario.

Be careful, though. It is easy to go overboard in the refactoring and make your code a spaghetti mess that no one (including you) can understand. Unless performance is really that critical and cannot be obtained any other way, you should not carry it that far. Make it asynchronous and independent, but make it clear.

If multiple threads are just reading from a variable (and there is no chance at all of any thread writing to it), then no synchronization is necessary. All threads can have unrestricted access. This is automatically the case for immutable objects like strings or immutable value types, but can be the case for any type of object if you ensure its immutability while multiple threads are reading it.

If you do have multiple threads writing to some shared variable, see if you can remove the need for synchronized access by converting usage to a local variable. If you can create a temporary copy to act on, no synchronization is necessary. This is especially important for repeated synchronized access. You will need to convert repeated access to a shared variable to repeated access of a local variable followed by a single shared access, as in the following simple example of multiple threads adding items to a shared collection.

```
object syncObj = new object();
var masterList = new List<long>();
const int NumTasks = 8;
Task[] tasks = new Task[NumTasks];

for (int i = 0; i < NumTasks; i++)
{
    tasks[i] = Task.Run(()=>
    {
        for (int j = 0; j < 5000000; j++)
        {
            lock (syncObj)
            {
                masterList.Add(j);
            }
        }
    });
}
Task.WaitAll(tasks);
```

This can be converted to the following:

```
object syncObj = new object();
var masterList = new List<long>();
const int NumTasks = 8;
Task[] tasks = new Task[NumTasks];
```

```
for (int i = 0; i < NumTasks; i++)
{
    tasks[i] = Task.Run(()=>
    {
        var localList = new List<long>();
        for (int j = 0; j < 5000000; j++)
        {
            localList.Add(j);
        }
        lock (syncObj)
        {
            masterList.AddRange(localList);
        }
    });
}
Task.WaitAll(tasks);
```

On my machine, the second code fragment takes less than half of the time as the first.

In the end, mutable shared state is a fundamentally performance-hostile pattern. It requires synchronization for data safety and performance begins to suffer. If, at all possible, your design can avoid locking, then you are close to implementing an ideal multithreaded system.

Synchronization Preference Order

If you decide you do need some sort of synchronization, then understand that not all of them have equal performance or behavior characteristics. Most situations call for just using a `lock` and that should usually be default. Using anything other than a `lock` should require some intense measurement to justify the additional complexity. In general, consider synchronization mechanisms in this order:

1. `lock/Monitor` class: Keep it simple, understandable, with a good balance of performance.

2. No synchronization at all: Remove shared mutable state, refactor, and optimize. It is harder, but if you can do it, it will usually perform better than using a lock (barring mistakes or detrimental design changes).
3. Simple `Interlocked` methods: They can be more appropriate in some scenarios, but once the situation becomes more complex, turn to a `lock`.

And finally, if you can truly prove they are beneficial, then use fancier, more complex locks, but be warned: they are rarely as useful as you might think.

- Asynchronous locks (see later in this chapter)
- Everything else

Specific circumstances may dictate or preclude the use of some of these techniques. For example, combining multiple `Interlocked` methods is not likely to outperform a single `lock` statement.

Memory Models

Before discussing the details of thread synchronization, we must take a short detour into the world of memory models. The memory model is the set of the rules in a system (hardware or software) that govern how read and write operations can be reordered by the compiler or by the processor across multiple threads. You cannot actually change anything about the model, but understanding it is critical to having correct code in all situations.

A memory model is described as “strong” if it has an absolute restriction on reordering, which prevents the compiler and hardware from doing many optimizations. A “weak” model allows the compiler and processor much more freedom to reorder read and write instructions in order to get potentially better performance. Most platforms fall somewhere between absolutely strong and completely weak.

The ECMA standard defines the minimum set of rules that the CLR must follow. It defines a very weak memory model, but a given implementation of the CLR may actually have a stronger model in place on top of it.

A specific processor architecture may also enforce a stronger memory model. For example, the x86/x64 architecture has a relatively strong memory model which will automatically prevent certain kinds of instruction reordering, but allow others. On the other hand, the ARM architecture has a relatively weak memory model. It is the JIT compiler's responsibility to ensure that not only are machine instructions emitted in the proper order, but that special instructions are used to ensure that the processor itself does not reorder instructions in a way that violates the CLR's memory model. These instructions may not be cheap to execute, which is another reason why avoiding synchronization at all is always preferable.

The practical difference between x86/x64 and ARM architectures has significant impact on your code, particularly if you have bugs in thread synchronization. Because the JIT compiler under ARM is more free to reorder reads and writes than it is under x86/x64, certain classes of synchronization bugs can go completely undetected on x86/x64 and only become apparent when ported to ARM.

In some cases, the CLR will help cover these differences for compatibility's sake, but it is good practice to ensure that the code is correct for the weakest memory model in use. The biggest requirement here is the correct use of `volatile` on state that is shared between threads. The `volatile` keyword is a signal to the JIT compiler that ordering matters for this variable. On x86/x64 it will enforce instruction ordering rules, but on ARM there will also be extra instructions emitted to enforce the correct semantics in the hardware. If you neglect `volatile`, the ECMA standard allows these instructions to be completely reordered and bugs could surface.

Other ways to ensure proper ordering of shared state is to use `Interlocked` or keep all accesses inside a full `lock`.

All of these methods of synchronization create what is called a memory barrier. Any reads that occur before that place in code may not be reordered after the barrier and any writes may not be reordered to be before the barrier. In this way, updates to variables are done correctly and seen by all CPUs.

Use volatile When Necessary

Consider the following classic example of an incorrect double-checked locking implementation that attempts to efficiently protect against multiple threads calling `DoCompletionWork`. It tries to avoid potentially expensive contentious calls to lock in the common case that there is no need to call `DoCompletionWork`.

```
private bool isComplete = false;
private object syncObj= new object();

// Incorrect implementation!
private void Complete()
{
    if (!isComplete)
    {
        lock (syncObj)
        {
            if (!isComplete)
            {
                DoCompletionWork();
                isComplete = true;
            }
        }
    }
}
```

While the `lock` statement will effectively protect its inner block, the outer check of `isComplete` is simultaneously accessed by multiple threads with no protection. Unfortunately, updates to this variable may be out of order due to compiler optimizations allowed under the memory model, leading to multiple threads seeing a false value even after another thread has set it to `true`. It is actually worse than that, though: It is possible that `isComplete` could be set to `true` before `DoCompletionWork()` completes, which means the program could be in an invalid state if other threads are checking and acting on the value of `isComplete`. Why not always use a `lock` around any access to `isComplete`? You could do that, but this leads to higher contention and is a more expensive solution than is strictly necessary.

To fix this, you need to instruct the compiler to ensure that accesses to this variable are done in the right order. You do this with the `volatile` keyword. The only change you need is:

```
private volatile bool isComplete = false;
```

To be clear, `volatile` is for program correctness, not performance. In most cases it does not noticeably help or hurt performance. If you can use it, it is better than using a `lock` in any high-contention scenario, which is why the double-checked locking pattern is useful.

Double-checked locking is often used for the singleton pattern when you want the first thread that uses the value to initialize it. This pattern is encapsulated in .NET by using the `Lazy<T>` class, which internally uses the double-checked locking pattern. You should prefer to use `Lazy< T >` rather than implement your own pattern. See Chapter 6 for more information about using `Lazy< T >`.

Use Monitor (lock)

The simplest way to protect any code region is with the `Monitor` object, which in C# has a keyword equivalent.

This code:

```
object obj = new object();
bool taken = false;
try
{
    Monitor.Enter(obj, ref taken);
}
finally
{
    if (taken)
    {
        Monitor.Exit(obj);
    }
}
```

is equivalent to this:

```
object obj = new object();

lock(obj)
{
    ...
}
```

The `taken` parameter is set to `true` if no exceptions occurred. This is guaranteed so that you can correctly call `Exit`.

In general, you should prefer using `Monitor/lock` versus other more complicated locking mechanisms until proven otherwise. `Monitor` is a hybrid lock in that it first tries to spin in a loop for a while before it enters a wait state and gives up the thread. This makes it ideal for places where you anticipate little or short-lived contention.

`Monitor` also has a more flexible API that can be useful if you have optional work to do if you cannot acquire the lock immediately.

```
object obj = new object();
bool taken = false;
try
{
    Monitor.TryEnter(obj, ref taken);
    if (taken) // do work that needs the lock
    {...}
    else       // do something else
    {...}
}
finally
{
    if (taken)
    {
        Monitor.Exit(obj);
    }
}
```

In this case, `TryEnter` returns immediately, regardless of whether it got the lock. You can test the `taken` variable to know what to do. There are also overloads that accept a timeout value.

Which Object To Lock On?

The `Monitor` class takes a synchronization object as its argument. You can pass any reference type object into this method. Reference types are required because, unlike value types, each reference type object contains a sync block as part of its memory layout. You can read more about object layout in Chapter 5. Care should be taken in selecting which object to use. If you pass in a publicly visible object, there is a possibility that another piece of code could also use it as a synchronization object, even if the synchronization is not needed between those two sections of code. If you pass in a complex object, you run the risk of functionality in that class taking a lock on itself. Both of these situations can lead to poor performance or worse: deadlocks.

To avoid this, it is almost always wise to allocate a plain, private object specifically for your locking purposes, as in the examples above.

On the other hand, I have seen situations where explicit synchronization objects can lead to problems, particularly when there were an enormous number of objects and the overhead of an extra field in the class was burdensome. In this case, you can find some other safe object that can serve as a synchronization object, or better, refactor your code to not need the lock in the first place.

There are some classes of objects you should never use as a sync object for `Monitor`. These include any `MarshalByRefObject` (a proxy object that will not protect the underlying resource), `string` objects (which are interned and shared unpredictably), `System.Type`, or a value type (which will be boxed every time you lock on it, prohibiting any synchronization from happening at all).

Ensure that you are not over-sharing the synchronization objects among different scenarios of locking. You want to ensure that only scenarios that conflict with each other use the same synchronization object. If you have multiple, independent scenarios that do not interfere with each other, do not reuse the same object, but use two different lock objects to ensure better scalability.

Lock Scope

When you do implement synchronization around some code, you have to decide how much code to wrap inside of it. In general, the answer should be: as little as possible (usually). Get in, do work, get out.

Less code covered by a lock means less opportunity for lock contention. However, this can be taken to extremes. If you find yourself repeatedly, in the same thread, taking a lock and releasing it, the overhead of the actual locking mechanism may in fact be more than the work you are doing inside the lock. This can easily happen in two types of common scenarios: collection access and loops.

For collections, consider the access pattern. Is it more efficient to take a lock on each individual access? Or is it better to take a lock at a higher level, do all the accesses, then release?

The same thing is true for loops. If you take a lock on each iteration of a loop, measure and see if it is better to just lock around the entire loop.

Use Interlocked Methods

Consider this code with a lock that is guaranteeing that only a single thread can execute the `Complete` method:

```
private bool isComplete = false;
private object completeLock = new object();

private void Complete()
{
    lock(completeLock)
    {
        if (isComplete)
        {
            return;
        }

        DoCompletionWorkHere();
    }

    isComplete = true;
}
```

```

    }
}
```

You need two fields and a few lines of code to determine if you should even enter the method, which seems wasteful. Instead, call the `Interlocked.Increment` method:

```

private int isComplete = 0;

private void Complete()
{
    if (Interlocked.Increment(ref isComplete) == 1)
    {
        DoCompletionWorkHere();
    }
}
```

Or consider a slightly different situation where `Complete` may be called multiple times, but you want only to enter it based on some internal state, and then enter it only once.

```

enum State { Executing, Done };
private int state = (int)State.Executing;

private void Complete()
{
    if (Interlocked.CompareExchange (ref state, (int)State.Done,
                                    (int)State.Executing) == (int)State.Executing)
    {
        DoCompletionWorkHere();
    }
}
```

The first time `Complete` executes, it will compare the `state` variable against `State.Executing` and if they are equal, replace `state` with the value `State.Done`. The next thread that executes this will compare `state` against `State.Executing`, which will not be true, and `CompareExchange` will return `State.Done`, failing the `if` statement.

`Interlocked` methods translate into a single processor instruction and are atomic. They are perfect for this kind of simple synchronization. There are a number of `Interlocked` methods you can use for simple synchronization, all of which perform the operation atomically:

- `Add`: Adds two integers, replacing the first one with the sum and returning the sum.
- `CompareExchange`: Accepts values A, B, and C. Compares values A and C, and, if equal, replaces A with B, and returns original value. See the `LockFreeStack` example below.
- `Increment`: Adds one to the value and returns the new value.
- `Decrement`: Subtracts one from the value and returns the new value.
- `Exchange`: Sets a variable to a specified value and returns the original value.

All of these methods have multiple overloads for various data types.

`Interlocked` operations are memory barriers, thus they are not as fast as a simple write in a non-contention scenario. In general, you should not count on them to be significantly faster than a simple `lock` scenario, but you can see the `InterlockedVsLock` sample project for a simple benchmark that shows a very slight advantage.

As simple as they are, `Interlocked` methods can implement more powerful concepts such as lock-free data structures. But a serious word of caution: Be very careful when implementing your own data structures like this. The seductive words “lock-free” can be misleading. They are really a synonym for “repeat the operation until correct.” Once you start having multiple calls to `Interlocked` methods, it is extremely likely to be less efficient than just using `lock` in the first place, even assuming your code is 100% correct (which it likely is not). It can be very difficult to get it right or performant. Implementing data structure like this is great for education, but in real code your default choice should always be to use the built-in .NET collections. If you do implement your own

thread-safe collection, take great pains to ensure that it is not only 100% functionally correct, but that it performs better than the existing .NET choices. If you use `Interlocked` methods, ensure that they provide more benefit than a simple `lock`. There are very, very few situations in which you will need to do this kind of work.

As an example (from the accompanying source in the `LockFreeStack` project), here is a simple implementation of a stack that uses `Interlocked` methods to maintain thread safety without using any of the heavier locking mechanisms.

```
class LockFreeStack<T>
{
    private class Node
    {
        public T Value;
        public Node Next;
    }

    private Node head;

    public void Push(T value)
    {
        var newNode = new Node() { Value = value };

        while (true)
        {
            newNode.Next = this.head;
            if (Interlocked.CompareExchange(ref this.head,
                newNode,
                newNode.Next)
                == newNode.Next)
            {
                return;
            }
        }
    }

    public T Pop()
    {
        while (true)
        {
            Node node = this.head;
```

```
    if (node == null)
    {
        return default(T);
    }
    if (Interlocked.CompareExchange(ref this.head,
                                    node.Next,
                                    node)
        == node)
    {
        return node.Value;
    }
}
}
```

This code fragment demonstrates a common pattern with implementing data structures or more complex logic using `Interlocked` methods: looping. Often, the code will loop, continually testing the results of the operation until it succeeds. In most scenarios, only a handful of iterations will be performed.

While `Interlocked` methods are simple and relatively fast, you will often need to protect more substantial areas of code and they will fall short (or at least be unwieldy and complex).

Asynchronous Locks

Starting in .NET 4.5, there is some interesting functionality that may expand to other types in the future. The `SemaphoreSlim` class has a `WaitAsync` method that returns a `Task`. Instead of blocking on a wait, you can schedule a continuation on the `Task` that will execute once the semaphore permits it to run. There is no entry into kernel mode and no blocking. When the lock is no longer contended, the continuation will be scheduled like a normal `Task` on the thread pool. Usage is the same as for any other `Task`.

To see how it works, consider first an example using the standard, blocking, wait mechanisms. This sample code (from the `WaitAsync` project in the sample code) is rather trivial, but demonstrates how the threads hand off control via the semaphore.

```
class Program
{
    static SemaphoreSlim semaphore = new SemaphoreSlim(1);
    const int WaitTimeMs = 1000;

    static void Main(string[] args)
    {
        Task.Run((Action)Func1);
        Task.Run((Action)Func2);
        Console.ReadKey();
    }

    static void Func1()
    {
        while (true)
        {
            semaphore.Wait();
            Console.WriteLine("Func1");
            semaphore.Release();
            Thread.Sleep(WaitTimeMs);
        }
    }

    static void Func2()
    {
        while (true)
        {
            semaphore.Wait();
            Console.WriteLine("Func2");
            semaphore.Release();
            Thread.Sleep(WaitTimeMs);
        }
    }
}
```

Each thread loops indefinitely while waiting for the other thread to finish its loop operation. When the `Wait` method is called, that thread will block until the semaphore is released. In a high-throughput program, that blocking is extremely wasteful and can reduce processing capacity and increase the size of the thread pool. If the block lasts long enough, a kernel transition may occur, which is yet more wasted time.

To use `WaitAsync`, replace the methods with these implementations:

```
static void AsyncFunc1()
{
    semaphore.WaitAsync().ContinueWith(_ =>
    {
        Console.WriteLine("AsyncFunc1");
        semaphore.Release();
        Thread.Sleep(WaitTimeMs);
    }).ContinueWith(_ => AsyncFunc1());
}

static void AsyncFunc2()
{
    semaphore.WaitAsync().ContinueWith(_ =>
    {
        Console.WriteLine("AsyncFunc2");
        semaphore.Release();
        Thread.Sleep(WaitTimeMs);
    }).ContinueWith(_ => AsyncFunc2());
}
```

Note that the loop was removed in lieu of a chain of continuations that will call back into these methods. It is logically recursive, but not actually so because each continuation starts the stack fresh with a new `Task`.

No blocking occurs with `WaitAsync`, but this is not free functionality. There is still overhead from increased `Task` scheduling and function calls. If your program schedules a lot of `Task` objects, the increased scheduling pressure may offset the gains from avoiding the blocking call. If your locks are extremely short-lived (they just spin and never enter kernel mode) or rare, then it may be better to just block for the few microseconds it takes to get through a standard lock. On the other hand, if you need to execute a relatively long-running task under a lock, this may be the better choice because the overhead of a new `Task` is less than the time you will block the processor otherwise. You will need to carefully measure and experiment.

If this pattern is interesting to you, see Stephen Toub's series of articles on asynchronous coordination primitives, titled Building Async Coordination Primitives (Parts 1–7), which cover more types and patterns.

Other Locking Mechanisms

There are many locking mechanisms you can use, but you should prefer to stick with as few as you can get away with. As in many things, but especially multi-threading, the simpler the better.

The simplest way to ensure that a method can only ever be called by a single thread is to decorate it with the `MethodImplOptions.Synchronized` option. You can apply it to static or instance methods.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public void DoSomething(int arg)
{
    ...
}
```

However, this is a very blunt tool and you should rarely need to use something so coarse. It is usually better to lock on a narrower scope from within the method.

If you know that a lock is extremely short-lived (tens of cycles) and want to guarantee that it never enters a wait state, then you can use a `SpinLock`. It will just spin a loop until the contention is done. In most cases, `Monitor`, which spins first, then enters a wait if necessary, is a better default choice.

```
private SpinLock spinLock = new SpinLock();

private void DoWork()
{
    bool taken = false;
    try
    {
        spinLock.Enter(ref taken);
        // Do some work
    }
    finally
    {
        if (taken)
        {
            spinLock.Exit();
        }
    }
}
```

```
}
```

In general, avoid other locking mechanisms if you can. They are usually not nearly as performant as the simple `Monitor`. Objects like `ReaderWriterLockSlim`, `Semaphore`, `Mutex`, or other custom synchronization objects definitely have their place, but they are often complex and error-prone.

Completely avoid `ReaderWriterLock`—it is deprecated and should never be used for any reason.

If there is a `-Slim` version of a synchronization object, prefer it to the non-`Slim` version. The `-Slim` versions are all hybrid locks which means they implement some form of spinning before they enter a kernel transition, which is much slower. `-Slim` locks are much better when the contention is expected to be low and brief.

For example, both `ManualResetEvent` and `ManualResetEventSlim` classes exist, and there is an `AutoResetEvent` class, but there is no `AutoResetEventSlim` class. However, you can use `SemaphoreSlim` with the `initialCount` parameter set to one for the same effect.

Concurrency and Collections

There are a handful of collections provided with .NET that allow concurrent access from multiple threads. They are all in the `System.Collections.Concurrent` namespace and include:

- `ConcurrentBag<T>`: unordered collection
- `ConcurrentDictionary< TKey , TValue >`: key/value pairs
- `ConcurrentQueue<T>`: first-in/first-out queue
- `ConcurrentStack<T>`: last-in/first-out stack

Most of these are implemented internally using `Interlocked` or `Monitor` synchronization primitives and I encourage you to examine their implementations using an IL reflection tool.

They are convenient, but you need to be careful – every single access to the collection involves synchronization. Often, that is overkill and could harm your performance if there is a lot of contention. If you have necessarily “chatty” read/write access patterns to the collection, this low-level synchronization may make sense.

This section will cover a few alternatives to concurrent collections that may simplify your locking requirements. See Chapter 6 for a discussion of collections in general, including those listed here, particularly for a description of the APIs unique to these collections, which can be tricky to get right.

Lock At a Higher Level

If you find yourself updating or reading many values at once, you should probably use a non-concurrent collection and handle the locking yourself at a higher level (or find a way to not need synchronization at all – see the next section for one idea).

The granularity of your synchronization mechanism has an enormous impact on overall efficiency. In many cases, making batch updates under a single lock is better than taking a lock for every single small update. In my own informal testing, I found that inserting an item into a `ConcurrentDictionary< TKey, TValue >` is about 2x slower than with a standard `Dictionary< TKey, TValue >`.

You will have to measure and judge the trade-off in your application.

Also note that sometimes you need to lock at a higher level to ensure that context-specific constraints are honored. Consider the classic example of bank transfer between two accounts. The balance of both accounts must be changed individually, of course, but the transaction only makes sense when considered together. Database transactions are a related concept. A single row insertion by itself may be atomic, but to guarantee integrity of your operations, you may need to utilize transactions to ensure higher-level atomicity.

Replace an Entire Collection

If your data is mostly read-only then you can safely use a non-concurrent collection when accessing it. When it is time to update the collection, you can generate a new collection object entirely and just replace the original reference once it is loaded, as in the following example:

```
private volatile Dictionary<string, MyComplexObject> data = new
    Dictionary<string, MyComplexObject>();

public Dictionary<string, MyComplexObject> Data
{
    get
    {
        return data;
    }
}

private void UpdateData()
{
    var newData = new Dictionary<string, MyComplexObject>();
    newData["Foo"] = new MyComplexObject();
    ...
    data = newData;
}
```

Notice the `volatile` keyword which ensures that `data` will not be updated until `newData` is fully constructed, and that when it is updated, it will be valid in all threads.

If a consumer of this class needed to access the `Data` property multiple times and did not want this object swapped out from under it to a new one, it can create a local copy of the reference and use that instead of the original property.

```
private void CreateReport(DataSource source)
{
    Dictionary<string, MyComplexObject> data = source.Data;

    foreach(var kvp in data)
    {
```

```
    ...
}
```

This is not a foolproof strategy. It makes your code a little more complex and spreads around the responsibility of correct data structure usage. The rest of your program may need to be aware of special semantics around using this data structure. You also need to balance this replacement against the cost of possibly incurring a full garbage collection when the new collection is allocated. As long as the reload is a rare occurrence and you can handle the occasional full GC, this may be the right pattern for many scenarios. See Chapter 2 for more information on avoiding full GCs.

Copy Your Resource Per-Thread

If you have a resource that is lightweight, not thread-safe, and is used a lot on multiple threads, you could consider wrapping it in a `ThreadLocal<T>` object.

Here is a classic example, using the `Random` class, which is not thread-safe.

```
private static readonly ThreadLocal<Random> threadLocalRand
    = new ThreadLocal<Random>(()=>new Random());

static void Main(string[] args)
{
    int[] results = new int[100];

    Parallel.For(0, 5000,
        i =>
    {
        var randomNumber = threadLocalRand.Value.Next(100);
        Interlocked.Increment(ref results[randomNumber]);
    });
}
```

The `ThreadLocal<T>` constructor takes a `Func<T>` to allow you to specify a factory method. This will be called the first time the variable is used on a thread, before the object is returned to you for usage.

`ThreadLocal<T>` is available in .Net 4 and higher. If for some reason, you need to use an earlier version of .Net, there is an alternative: the `[ThreadStatic]` attribute. It can only be applied to `static` variables.

Here is the same example implemented using `[ThreadStatic]`. This example comes from the MultiThreadRand sample project:

```
[ThreadStatic]
static Random threadStaticRand;

static void Main(string[] args)
{
    int[] results = new int[100];

    Parallel.For(0, 5000,
        i =>
    {
        // thread statics are not initialized
        if (threadStaticRand == null)
        {
            threadStaticRand = new Random();
        }
        var randomNumber = threadStaticRand.Next(100);
        Interlocked.Increment(ref results[randomNumber]);
    });
}
```

You should always assume that statics marked as `[ThreadStatic]` are not initialized at the time of first use .NET will only initialize the first one. The rest will have default values (usually `null`). Because of its limitations, there is little reason to use `[ThreadStatic]` and you should prefer `ThreadLocal<T>` for most circumstances.

It is important to note that `[ThreadStatic]` and `ThreadLocal<T>` are slower than non-thread-aware variables. You should prefer standard variables whenever possible, but consider these two possibilities as a simple way to convert a shared resource (needing a lock) into a non-shared resource (needing no synchronization).

Investigating Threads and Contention

One of the most difficult types of debugging to do is any issue relating to multiple threads. Spending the effort to get the code right in the first place pays huge dividends later as a lack of time spent debugging.

On the other hand, finding sources of contention in .NET is trivially easy and there are some nice advanced tools out there that can help with some general multithreading analysis.

Performance Counters

In the Process category exists the Thread Count counter.

In the Synchronization category, you can find the following counters:

- Spinlock Acquires/sec
- Spinlock Contentions/sec
- Spinlock Spins/sec

Under System, you can find the Context Switches/sec counter. It is difficult to know what the ideal value of this counter should be. You can find many conflicting opinions about this (I have commonly seen a value of 300 being normal, with 1,000 being too high), so I believe you should view this counter mostly in a relative sense and track how it changes over time, with large increases potentially indicating problems in your app.

.NET provides a number of counters in the .NET CLR LocksAndThreads category, including:

- **# of current logical Threads:** The number of managed threads in the process.

- **# of current physical Threads:** The number of OS threads assigned to the process to execute the managed threads, excluding those used only by the CLR.
- **Contention Rate /sec:** Important for detecting “hot” locks that you should refactor or remove.
- **Current Queue Length:** The number of threads blocked on a lock.

ETW Events

Of the following events, ContentionStart_V1 and ContentionStop are the most useful. The others may be interesting in a situation where the concurrency level is changing in unexpected ways and you need insights into how the thread pool is behaving.

- **ContentionStart_V1:** Contention has started. For hybrid locks, this does not count the spinning phase, but only when it enters an actual blocked state. Fields include:
 - Flags: 0 for managed, 1 for native.
- **ContentionStop:** Contention has ended.
- **ThreadPoolWorkerThreadStart:** A thread pool thread has started. Fields include:
 - ActiveWorkerThreadCount: Number of worker threads available, both those processing work and those waiting for work.
 - RetiredWorkerThreadCount: Number of worker threads being held in reserve if more threads are needed later.
- **ThreadPoolWorkerThreadStop:** A thread pool thread has ended. Fields are same as for ThreadPoolWorkerThreadStart.
- **IOThreadCreate_V1:** An I/O thread has been created. Fields include:

- Count: Number of I/O threads in pool.

See <https://docs.microsoft.com/dotnet/framework/performance/thread-pool-etw-events> for more information about thread-related ETW events.

Get Thread Information

Most debuggers will show you the running threads and let you selectively pause and unpause them (or as Visual Studio calls it, freeze and thaw).

In WinDbg, you can get detailed information about managed threads with the `!Threads` and the `!ThreadState` commands:

```
0:007> !Threads
```

```
ThreadCount:      3
UnstartedThread:  0
BackgroundThread: 2
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no
```

ID	OSID	ThreadOBJ	State	GC Mode	GC Alloc	Context	Domain	...
0	1	5580	034578a0	2a020	Preemptive	058ECED4:00000000	0344d1e0...	
5	2	4bb8	034675c8	2b220	Preemptive	00000000:00000000	0344d1e0...	
6	3	42b0	03486658	21220	Preemptive	00000000:00000000	0344d1e0...	

```
0:007> !ThreadState 2a020
```

```
Legal to Join
CoInitialized
In Multi Threaded Apartment
Fully initialized
```

The command `!ThreadState -special` will show you CLR-owned threads and what they are doing:

```
0:000> !Threads -special
ThreadCount:      3
UnstartedThread:  0
BackgroundThread: 2
PendingThread:    0
DeadThread:       0
Hosted Runtime:   no

ID OSID ThreadOBJ State GC Mode      GC Alloc Context Domain ...
0 1 5580 034578a0 2a020 Preemptive  058ECED4:00000000 0344d1e0...
5 2 4bb8 034675c8 2b220 Preemptive  00000000:00000000 0344d1e0...
6 3 42b0 03486658 21220 Preemptive  00000000:00000000 0344d1e0...

OSID Special thread type
4 4c30 DbgHelper
5 4bb8 Finalizer
6 42b0 GC
```

In this case, you can see the finalizer thread and a GC thread (presumably the background GC thread, since this application uses workstation GC).

Visualizing Tasks and Threads with Visual Studio

Visual Studio has an optional tool called the Concurrency Visualizer. This tool relies on ETW events, but shows them in a graphical form so that you can see exactly what all the `Task` objects and threads are doing in your application. It can tell you when `Task` delegates start and stop, whether a thread is blocked, doing CPU work, waiting on I/O, and a lot more.

It is important to note that capturing this type of information can significantly degrade the application's performance, as an event will be recorded every time a thread changes state, which is extremely frequent.

As of the time of this writing, the Concurrency Analyzer is not available for Visual Studio 2017, but you can use the more rudimentary Profiling Wizard to collect this data if needed. Launch the Profile Wizard (from the Analyze | Performance Profiler...) and select the concurrency option.

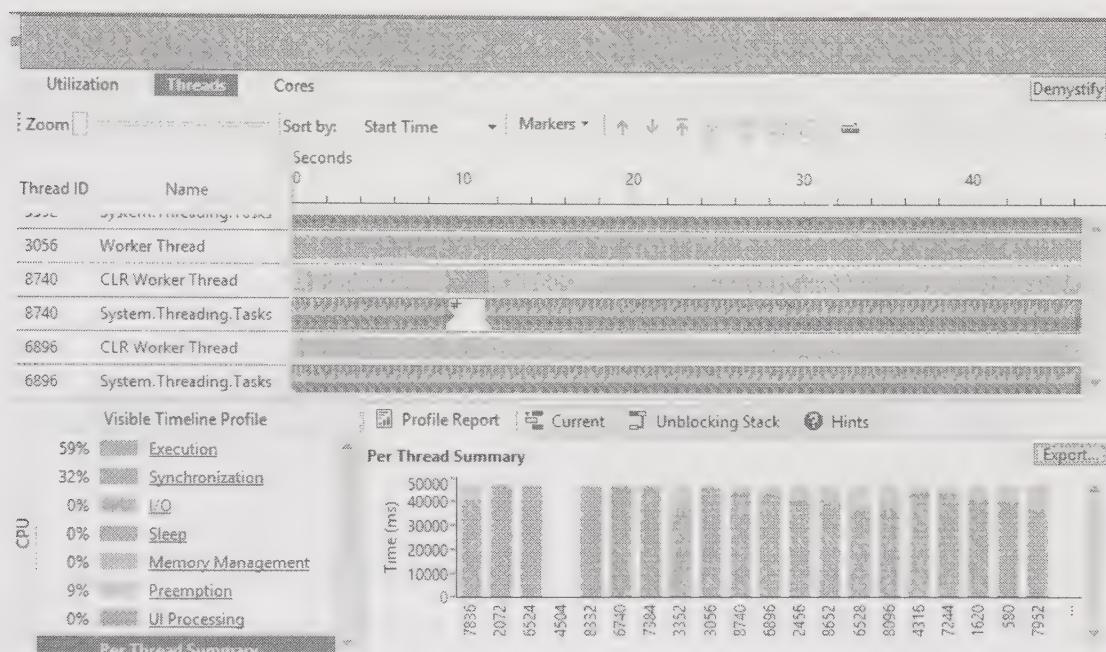


Figure 4.3. Visual Studio's Thread Times view combines threads, tasks, CPU time, blocked time, interrupts, and more into a single correlated timeline.

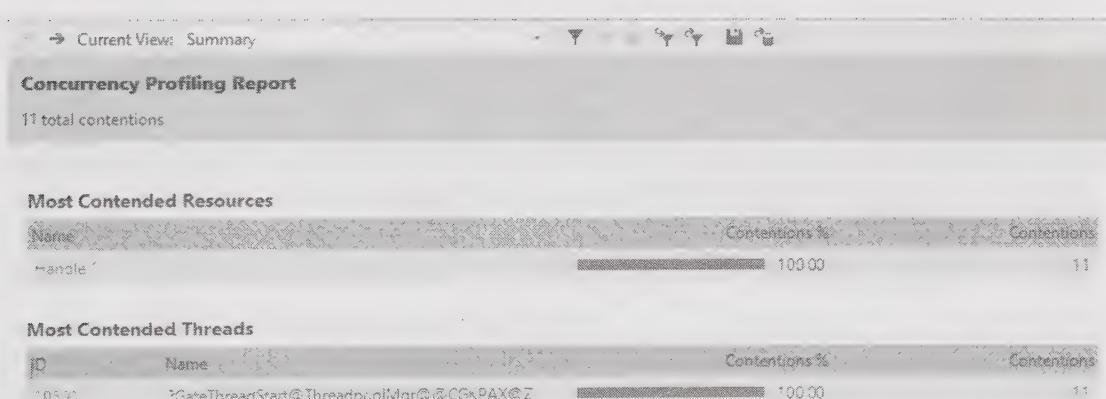


Figure 4.4. A rudimentary concurrency analyzer. This shows which handles and threads exhibit the most contention.

Using PerfView to find Lock Contention

A strong alternative to Visual Studio, especially for analysis of production workloads is PerfView. You can use the sample program HighContention to see this in action. Run the program and collect .NET events using PerfView. Once the .etl file is ready to view, open the Any Stacks view and look for an entry named “Event Microsoft-Windows-DotNETRuntime/Contention/Start” and double-click it. It will open a stack view of your process and show stacks that contribute to thread contention.

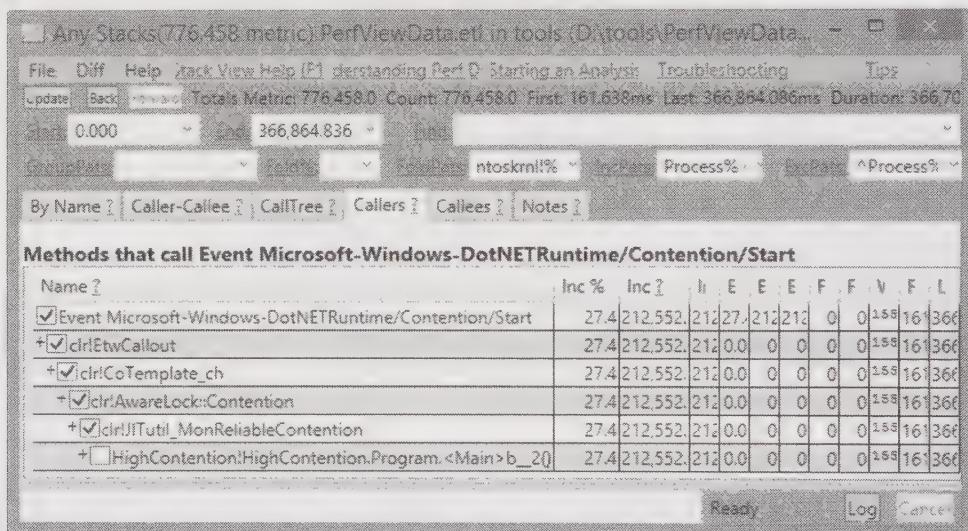


Figure 4.5. PerfView shows which stacks are leading to higher contention for all managed synchronization objects. In this case, you can see that the contention comes from an anonymous method inside `Main`.

Where Are My Threads Blocking On I/O?

You can use PerfView to collect information about threads such as when they enter various states. This gives you more accurate information about overall wall-clock time spent in your program, if it is not using the CPU. However, be aware that turning on this option slows down your program considerably.

In PerfView's Collection dialog, check the “Thread Times” box as well as the other defaults (Kernel, .NET, and CPU events).

Once collection is done, you will see a Thread Times node in the results tree. Open that and look at the By Name tab. You should see two main categories at the top: BLOCKED_TIME and CPU_TIME. Double-click BLOCKED_TIME to see the callers for that group.

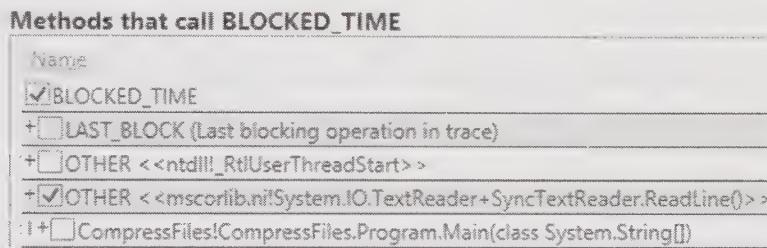


Figure 4.6. A *PerfView* stack of blocked time, caused by a call to the *TextReader.ReadLine* method.

Summary

Use asynchronous code such as `async` and `await` (or `Task` objects directly) when you need to avoid blocking the UI thread, parallelize work on multiple CPUs, or avoid wasting CPU capacity by blocking threads waiting for I/O to complete. Use `Task` objects instead of pure threads. Do not wait on `Task` objects, but schedule a continuation to be executed when the `Task` is complete. Simplify `Task` syntax with `async` and `await`. Consider using TPL Dataflow for appropriate processing workflows.

Never block on I/O. Always use asynchronous APIs when reading from or writing to `Stream` objects. Use continuations to schedule a callback for when the I/O completes.

Try to avoid locking if at all possible, even if you need to significantly restructure your code. When needed, use the simplest locking possible. For short or rarely contended sections, use `lock/Monitor` and use a private field as the syn-

chronization object. For simple state changes, use `Interlocked` methods. If you have a highly contended critical section that lasts more than a few milliseconds, consider using the asynchronous locking pattern like `SemaphoreSlim`.

Chapter 5

General Coding and Class Design

This chapter covers general coding and type design principles not covered elsewhere in this book. .NET contains features for many scenarios and while many of them are at worst performance-neutral, some are decidedly harmful to good performance. You must decide what the right approach in a given situation is.

If I were to summarize a single principle that will show up throughout this chapter and the next, it is:

In-depth performance optimization will often defy code abstractions.

This means that when trying to achieve extremely good performance, you will need to understand and possibly rely on the implementation details at all layers. Many of those are described in this chapter.

Classes and Structs

Instances of a class are always allocated on the heap and accessed via a pointer dereference. Passing them around is cheap because it is just a copy of the pointer (4 or 8 bytes). However, an object also has some fixed overhead: 8 bytes for

32-bit processes and 16 bytes for 64-bit processes. This overhead includes the pointer to the method table and a sync block field that is used for multiple purposes. However, if you examine an object that had no fields in the debugger, you will see that the size is reported as 12 bytes (32-bit) or 24 bytes (64-bit). Why is that? .NET will align all objects in memory and these are the effective minimum object sizes.

A struct (also known as a value type) has no overhead at all and its memory usage is a sum of the size of all its fields. If a struct is declared as a local variable in a method, then the struct is allocated on the stack. If the struct is declared as part of a class, then the struct's memory will be part of that class's memory layout (and thus exist on the heap). When you pass a struct to a method it is copied byte for byte. Because it is not on the heap, allocating a struct will never cause a garbage collection. However, if you start allocating large structs all the time, you may start running into stack space limitations if you have very deep stacks (which is very possible with some frameworks).

There is thus a tradeoff here. You can find various pieces of advice about the maximum recommended size of a struct, but I would not get caught up on the exact number. In most cases, you will want to keep struct sizes very small, especially if they are passed around, but you can also pass structs by reference so the size may not be an important issue to you. The only way to know for sure whether it benefits you is to consider your usage pattern and do your own profiling.

There is a huge difference in efficiency in some cases. While the overhead of an object might not seem like very much, consider an array of objects and compare it to an array of structs. Assume the data structure contains 16 bytes of data, the array length is 1,000,000, and this is a 32-bit system.

For an array of objects the total space usage is:

$$\begin{aligned} & \text{8 bytes array overhead} \\ & + (4 \text{ byte pointer size} \times 1000000) \\ & + (\underline{(8 \text{ bytes overhead} + 16 \text{ bytes data}) \times 1000000}) \\ & \qquad \qquad \qquad = 28 \text{ MB} \end{aligned}$$

For an array of structs, the results are dramatically different:

$$\begin{array}{r} \text{8 bytes array overhead} \\ + (16 \text{ bytes data} \times 1000000) \\ \hline = 16 \text{ MB} \end{array}$$

With a 64-bit process, the object array takes over 40 MB while the struct array still requires only 16 MB.

As you can see, in an array of structs, the same size of data takes less memory. With the overhead of reference types, you are also inviting a higher rate of garbage collections just from the added memory pressure.

Aside from space, there is also the matter of CPU efficiency. CPUs have multiple levels of caches. Those closest to the processor are very small, but extremely fast and optimized for sequential access.

An array of structs has many sequential values in memory. Accessing an item in the struct array is very simple. Once the correct entry is found, the right value is there already. This can mean a huge difference in access times when iterating over a large array. If the value is already in the CPU's cache, it can be accessed an order of magnitude faster than if it were in RAM.

To access an item in the object array requires an access into the array's memory, then a dereference of that pointer to the item elsewhere in the heap. Iterating over object arrays dereferences an extra pointer, jumps around in the heap, and evicts the CPU's cache more often, potentially squandering more useful data.

This lack of overhead for both CPU and memory is a prime reason to favor structs in many circumstances – they can buy you significant performance gains when used intelligently because of the improved memory locality, lack of GC pressure, and, since structs naturally live on the stack, it encourages a programming model without shared mutable state. Because of these natural limits, you should strongly consider making all of your structs immutable. However, if you find yourself wanting to modify fields within a struct that is itself a property on another class, look at the `ref`-return functionality described later in this chapter. Using this new functionality in C#7, you can avoid the struct copies that would otherwise sink performance.

A Mutable struct Exception: Field Hierarchies

I mentioned earlier that structs should be kept small to avoid spending significant time copying them, but there are occasional uses for large, mutable structs: field hierarchies. Consider an object that tracks a lot of details of some commercial process, such as a lot of time stamps.

```
class Order
{
    public DateTime ReceivedTime {get;set;}
    public DateTime AcknowledgeTime {get;set;}
    public DateTime ProcessBeginTime {get;set;}
    public DateTime WarehouseReceiveTime {get;set;}
    public DateTime WarehouseRunnerReceiveTime {get;set;}
    public DateTime WarehouseRunnerCompletionTime {get;set;}
    public DateTime PackingBeginTime {get;set;}
    public DateTime PackingEndTime {get;set;}
    public DateTime LabelPrintTime {get;set;}
    public DateTime CarrierNotifyTime {get;set;}
    public DateTime ProcessEndTime {get;set;}
    public DateTime EmailSentToCustomerTime {get;set;}
    public DateTime CarrerPickupTime {get;set;}

    // lots of other data ...
}
```

To simplify your code, it would be nice to segregate all of those times into their own sub-structure, still accessible via the `Order` class via some code like this:

```
Order order = new Order();
Order.Times.ReceivedTime = DateTime.UtcNow;
```

You could put all of them into their own class.

```
class OrderTimes
{
    public DateTime ReceivedTime {get;set;}
    public DateTime AcknowledgeTime {get;set;}
    public DateTime ProcessBeginTime {get;set;}
    public DateTime WarehouseReceiveTime {get;set;}
```

```

public DateTime WarehouseRunnerReceiveTime {get;set;}
public DateTime WarehouseRunnerCompletionTime {get;set;}
public DateTime PackingBeginTime {get;set;}
public DateTime PackingEndTime {get;set;}
public DateTime LabelPrintTime {get;set;}
public DateTime CarrierNotifyTime {get;set;}
public DateTime ProcessEndTime {get;set;}
public DateTime EmailSentToCustomerTime {get;set;}
public DateTime CarrerPickupTime {get;set;}
}

class Order
{
    public OrderTimes Times;
}

```

However, this does introduce an additional 12 or 24-bytes of overhead for every `Order` object. If you need to pass the `OrderTimes` object as a whole to various methods, maybe this makes sense, but why not just pass the reference to the entire `Order` object itself? If you have thousands of `Order` objects being processed simultaneously, this can cause more garbage collections to be induced. It is also an extra memory dereference.

Instead, change `OrderTimes` to be a struct. Accessing the individual properties of the `OrderTimes` struct via a property on `Order` (`order.Times.ReceivedTime`) will not result in a copy of the struct (.NET optimizes that reasonable scenario). This way, the `OrderTimes` struct becomes part of the memory layout for the `Order` class almost exactly like it was with no substructure and you get to have better-looking code as well.

The trick here is to treat the fields of the `OrderTimes` struct just as if they were fields on the `Order` object. You do not need to pass around the `OrderTimes` struct as an entity in and of itself – it is just an organization mechanism.

Virtual Methods and Sealed Classes

Do not mark methods `virtual` by default, “just in case.” However, if `virtual` methods are necessary for a coherent design in your program, you probably should not go too far out of your way to remove them.

Making methods `virtual` prevents certain optimizations by the JIT compiler, notably the ability to inline them. Methods can only be inlined if the compiler knows 100% which method is going to be called. Marking a method as `virtual` removes this certainty, though there are other factors, covered in Chapter 3, that are perhaps more likely to invalidate this optimization.

Closely related to `virtual` methods is the notion of sealing a class, like this:

```
public sealed class MyClass {}
```

A class marked as `sealed` is declaring that no other classes can derive from it. In theory, the JIT could use this information to inline more aggressively, but it does not do so currently. Regardless, you should mark classes as `sealed` by default and not make methods `virtual` unless they need to be. This way, your code will be able to take advantage of any current as well as theoretical future improvements in the JIT compiler.

If you are writing a class library that is meant to be used in a wide variety of situations, especially outside of our organization, you need to be more careful. In that case, having `virtual` APIs may be more important than raw performance to ensure your library is sufficiently reusable and customizable. But for code that you change often and is used only internally, go the route of better performance.

Properties

Be careful with accessing properties. Properties look syntactically like fields, but underneath they are actually function calls. It is considered good manners to implement properties in as light-weight manner as possible, but if it were as simple and cheap as field access, then properties would not exist. They largely exist so that people can add validation and other extra functionality around accessing or modifying a field's value.

If the property access is in a loop, it is possible that the JIT will inline the call, but it is not guaranteed.

When in doubt, examine the code for the properties you are accessing in performance critical areas, and make your decisions accordingly.

Override Equals and GetHashCode for Structs

An important part of using structs is overriding the `Equals` and `GetHashCode` methods. If you do not, you will get the default versions, which are not at all good for performance. To get an idea of how bad it is, use an IL viewer and look at the code for the `ValueType.Equals` method. It involves reflection over all the fields in the struct. There is, however, an optimization for blittable types. A blittable type is one that has the same in-memory representation in managed and unmanaged code. They are limited to the primitive numeric types (such as `Int32`, `UInt64`, for example, but not `Decimal`, which is not a primitive) and `IntPtr/UIntPtr`. If a struct is comprised of all blittable types, then the `Equals` implementation can do the equivalent of byte-for-byte memory comparison across the whole struct. Otherwise, always implement your own `Equals` method.

If you just override `Equals(object other)`, then you are still going to have worse performance than necessary, because that method involves casting and boxing on value types. Instead, implement `Equals(T other)`, where `T` is the type of your struct. This is what the `IEquatable<T>` interface is for, and all structs should implement it. During compilation, the compiler will prefer the more strongly typed version whenever possible. The following code snippet shows you an example.

```
struct Vector : IEquatable<Vector>
{
    public int X { get; }
    public int Y { get; }
    public int Z { get; }

    public int Magnitude { get; }

    public Vector(int x, int y, int z, int magnitude)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
        this.Magnitude = magnitude;
    }
}
```

```
public override bool Equals(object obj)
{
    if (obj == null)
    {
        return false;
    }
    if (obj.GetType() != this.GetType())
    {
        return false;
    }
    return this.Equals((Vector)obj);
}

public bool Equals(Vector other)
{
    return this.X == other.X
        && this.Y == other.Y
        && this.Z == other.Z
        && this.Magnitude == other.Magnitude;
}

public override int GetHashCode()
{
    return X ^ Y ^ Z ^ Magnitude;
}
```

If a type implements `IEquatable<T>` .NET's generic collections will detect its presence and use it to perform more efficient searches and sorts.

You may also want to implement the `==` and `!=` operators on your value types and have them call the existing `Equals<T>` method.

All of these methods should be implemented as optimally as possible. They should have the minimal number of operations, no duplication, and no memory allocation. They will be called in many unforeseen circumstances. For large collections, they could be called millions of times per second. Also, `GetHashCode` is used in many collections to very quickly narrow down the range of items they need to check for equality. If the hash code calculation produces too many collisions, then the potentially more expensive `Equals` method will be called too frequently.

If your type is sortable, then you should also implement the `IComparable<T>` interface to allow the `Sort` method of some collection types to automatically use it.

Even if you never compare structs or put them in collections, I still encourage you to implement these methods. You will not always know how they will be used in the future, and the price of the methods is only a few minutes of your time and a few bytes of IL that will never even get JITted.

It is not as important to override `Equals` and `GetHashCode` on classes because by default they only calculate equality based on their object reference. As long as that is a reasonable assumption for your objects, you can leave them as the default implementation.

Thread Safety

Classes should rarely be thread-safe, unless there is some inherent reason they need to be. This is rare outside of collection classes, and as we will see when we discuss those, even then you have to consider the question carefully.

For most cases, synchronization should happen at a higher level and the class itself should be unaware. This provides the most flexibility in class reuse.

One exception is static classes. Since these only have global state, you should consider making these thread-safe by default unless you have reason not to.

To learn more about thread synchronization, see Chapter 4.

Tuples

The generic `System.Tuple` class can be used to create simple data structures without creating explicit, named classes. `Tuple` is a reference type, which means it has all the overhead associated with classes. Starting with .NET 4.7 and C# 7, there is a value type version of tuples, `System.ValueTuple`. This should be preferred in most cases, but use the same judgment for deciding between any reference or value type designs, as described earlier.

```
var tuple = new ValueTuple<int, string>(1, "Ben");
int id = tuple.Item1;
```

Along with the new type, you can use some new language syntax to declare tuples:

```
(int, string) tuple = (1, "Ben");
int id = tuple.Item1;
```

Instead of using the `Item` property names, you can now name them:

```
(int id, string name) tuple = (1, name: "Ben");
int id = tuple.id;
```

You can use this syntax as method return or parameter types – it is all equivalent to using `ValueTuple`, and if you look at these values in a debugger, you will not see the property names you may have used, but just `Item1`, `Item2`, etc.

Interface Dispatch

The first time you call a method through an interface, .NET has to figure out which type and method to make the call on. It will first make a call to a stub that finds the right method to call for the appropriate object implementing that interface. Once this happens a few times, the CLR will recognize that the same concrete type is always being called and this indirect call via the stub is reduced to a stub of just a handful of assembly instructions that makes a direct call to the correct method. This group of instructions is called a monomorphic stub because it knows how to call a method for a single type. This is ideal for situations where a call site always calls interface methods on the same type every time.

The monomorphic stub can also detect when it is wrong. If at some point the call site uses an object of a different type, then eventually the CLR will replace the stub with another monomorphic stub for the new type.

If the situation is even more complex with multiple types and less predictability (for example, you have an array of an interface type, but there are multiple concrete types in that array) then the stub will be changed to a polymorphic stub that uses a hash table to pick which method to call. The table lookup is fast, but not as fast as the monomorphic stub. Also, this hash table is severely bounded in size and if you have too many types, you might fall back to the generic type lookup code from the beginning. This can be very expensive.

The stubs are created per call-site; that is, wherever the methods are called. Each call-site is updated as needed, independently of one another.

If this becomes a concern for you, you have a couple of options:

1. Avoid calling these objects through the common interface
2. Pick your common base interface and replace it with an abstract base class instead

This type of problem is not common, but it can hit you if you have a huge type hierarchy, all implementing a common interface, and you call methods through that root interface. You would notice this as high, unexplainable CPU usage at the call site for these methods.

During the design of a large system, we knew we were going to have potentially thousands of types that would likely all descend from a common type. We knew there would be a couple of places where we would need to access them from the base type. Because we had someone on the team who understood the issues around interface dispatch with this magnitude of problem size, we chose to use an abstract base class rather than a root interface instead.

To learn more about interface dispatch see Vance Morrison's blog entry on the subject, titled, "Digging into interface calls in the .NET Framework: Stub-based dispatch."

Avoid Boxing

Boxing is the process of wrapping a value type such as a primitive or struct inside an object that lives on the heap so that it can be passed to methods that require object references. Unboxing is getting the original value back out again.

Boxing costs CPU time for object allocation, copying, and casting, but, more seriously, it results in more pressure on the GC heap. If you are careless about boxing, it can lead to a significant number of allocations, all of which the GC will have to handle.

Obvious boxing happens whenever you do things like the following:

```
int x = 32;
object o = x;
```

The IL looks like this:

```
IL_0001: ldc.i4.s 32
IL_0003: stloc.0
IL_0004: ldloc.0
IL_0005: box [mscorlib]System.Int32
IL_000a: stloc.1
```

This means that it is relatively easy to find most sources of boxing in your code—just use ILDASM to convert all of your IL to text and do a search.

A very common way of having accidental boxing is using APIs that take `object` or `object[]` as a parameter. The most well-known of these is `String.Format` or the old style collections which only store `object` references and should be avoided completely for this and other reasons (see Chapter 6).

Boxing can also occur when assigning a struct to an interface reference. For example:

```
interface INameable
{
    string Name { get; set; }
```

```

}

struct Foo : INameable
{
    public string Name { get; set; }
}

void TestBoxing()
{
    Foo foo = new Foo() { Name = "Bar" };
    // This boxes!
    INameable nameable = foo;
    ...
}

```

If you test this out for yourself, be aware that if you do not actually use the boxed variable then the compiler will optimize out the boxing instruction because it is never actually touched. As soon as you call a method or otherwise use the value then the boxing instruction will be present.

Another thing to be aware of when boxing occurs is the result of the following code:

```

int val = 13;
object boxedVal = val;
val = 14;

```

What is the value of `boxedVal` after this?

Boxing looks just like reference aliasing, but it instead copies the value and there is no longer any relationship between the two values. In this example, `val` changes value to 14, but `boxedVal` maintains its original value of 13.

You can sometimes catch boxing happening in a CPU profile, but many boxing calls are inlined so this is not a reliable method of finding it. What will show up in a CPU profile of excessive boxing is heavy memory allocation through `new`.

If you do have a lot of boxing of structs and find that you cannot get rid of it, you should probably just convert the struct to a class, which may end up being cheaper overall.

Finally, note that passing a value type by reference is not boxing. Examine the IL and you will see that no boxing occurs. The address of the value type is sent to the method.

ref returns and locals

C# 7 introduced some new language syntax that enables easier direct memory access in safe code. The same benefits could be achieved earlier, with pointer access to private fields in unsafe code, but the standard way of coding would usually result in copying values, as we will see later in this section. With `ref`-return, you can have the benefits of completely safe code, proper class abstraction, as well as the performance benefit of direct memory access.

As a simple example, consider a local `ref` to an existing value:

```
int value = 13;
ref int refValue = value;

refValue = 14;
```

After the last line, what is in `value`? It is 14 because `refValue` actually refers to `value`'s memory location.

This functionality can also be used to get a reference to a class's private data:

```
class Vector
{
    private int magnitude;
    public ref int Magnitude {
        get { ref return this.magnitude; } }
}

class Program
{
    void TestMagnitude()
    {
        Vector v = new Vector();
        ref int mag = ref v.Magnitude;
```

```

    mag = 3;

    int nonRefMag = v.Magnitude;
    mag = 4;

    Console.WriteLine($"mag: {mag}");
    Console.WriteLine($"nonRefMag: {nonRefMag}");
}

}

```

What is the output of this program?

4
3

The first assignment sets the underlying value. The assignment to `nonRefMag` is interesting. Despite `Magnitude` being a `ref`-return property, because it was not called via `ref`, `'nonMagRef` will just get a copy of the value, just as if `Magnitude` were a typical, non-`ref` property. Thus `nonRefMag` retains the value it originally received, despite the underlying class's memory being changed. Remember that how you call a method is as important as how the method is declared.

You can also use `ref` to refer to a specific array location. This example is a method that zeroes the middle position in an array. The non-`ref` way of doing it would look something like this:

```

private static void ZeroMiddleValue(int[] arr)
{
    int midIndex = GetMidIndex(arr);
    arr[midIndex] = 0;
}

private static int GetMidIndex(int[] arr)
{
    return arr.Length / 2;
}

```

The `ref` version looks very similar:

```
private static void RefZeroMiddleValue(int[] arr)
{
    ref int middle = ref GetRefToMiddle(arr);
    middle = 0;
}

private static ref int GetRefToMiddle(int[] arr)
{
    return ref arr[arr.Length / 2];
}
```

With `ref`-return functionality, you can do previously illegal operations like putting a method on the left-hand side of an assignment:

```
GetRefToMiddle(arr) = 0
```

Since `GetRefToMiddle` returns a reference, not a value, you can assign to it.

Looking at these simple examples of usage, you may be tempted to say that it looks unlikely that there is large performance gain. For small one-offs this is true. The gain will come from repeated reference to a single location in memory, avoiding array offset math, or avoiding copying values.

A more powerful example is using `ref`-return to avoid copying struct values when you cannot use an immutable struct. Consider the following definitions:

```
struct Point3d
{
    public double x;
    public double y;
    public double z;

    public string Name { get; set; }
}

class Vector
{
    private Point3d location;
    public Point3d Location { get; set; }
    public ref Point3d RefLocation
```

```

    { get { return ref this.location; } }

public int Magnitude { get; set; }
}

```

Suppose you want to change `location` to be the origin (0,0,0). Without `ref`-return, this would mean copying the struct via the `Location` property, setting its values to 0, then calling the setter to put it back, like this:

```

private static void SetVectorToOrigin(Vector vector)
{
    Point3d location = vector.Location;
    pt.x = 0;
    pt.y = 0;
    pt.z = 0;
    vector.Location = pt;
}

```

With `ref`-return you can circumvent this copying:

```

private static void RefSetVectorToOrigin(Vector vector)
{
    ref Point3d location = ref vector.RefLocation;
    location.x = 0;
    location.y = 0;
    location.z = 0;
}

```

The difference in efficiency will depend on the size of the struct – the bigger it is, the slower it will take to execute the non-`ref` version of this method.

The `RefReturn` project in the accompanying source code for this book contains a simple benchmark with the above code that has this output:

Benchmarks:

`SetVectorToOrigin`: 40ms
`RefSetVectorToOrigin`: 20ms

If I add just a few more fields to the struct, the difference becomes starker:

Benchmarks:

SetVectorToOrigin: 470ms
RefSetVectorToOrigin: 20ms

Digging into the assembly code, you can see that the inefficient version has instructions for copying as well as a method call:

```
02E005A8  push      esi
02E005A9  cmp       al,byte ptr [ecx+24h]
02E005AC  lea       esi,[ecx+24h]
02E005AF  mov       eax,dword ptr [esi+18h]
02E005B2  fldz
02E005B4  fldz
02E005B6  fldz
02E005B8  lea       esi,[ecx+24h]
02E005BB  fxch      st(2)
02E005BD  fstp      qword ptr [esi]
02E005BF  fstp      qword ptr [esi+8]
02E005C2  fstp      qword ptr [esi+10h]
02E005C5  lea       edx,[esi+18h]
02E005C8  call      72BDDCB8
02E005CD  pop       esi
02E005CE  ret
```

While the `ref-return` version contains little more than value setting and, as a bonus, is inlined:

```
02E005E0  cmp       byte ptr [ecx],al
02E005E2  lea       eax,[ecx+8]
02E005E5  fldz
02E005E7  fstp      qword ptr [eax]
02E005E9  fldz
02E005EB  fstp      qword ptr [eax+8]
02E005EE  fldz
02E005F0  fstp      qword ptr [eax+10h]
02E005F3  ret
```

There are strict rules for when `ref`-return functionality can be used:

- You cannot assign the result of a regular (i.e., non-`ref`-return) method return value to a `ref` local variable. (However, `ref`-return values can be implicitly copied into non-`ref` variables.)
- You cannot return a `ref` of a local variable. The actual memory must persist beyond the local scope to avoid invalid memory access.
- You cannot reassign a `ref` variable to a new memory location after initialization.
- Struct methods cannot `ref`-return instance fields.
- You cannot use this functionality with `async` methods.

You likely will not frequently use this feature, but it is there when you need it, especially for the situations I described:

- Modifying fields in a property-exposed struct.
- Directly accessing an array location.
- Repeated access to the same memory location.

for vs. foreach

The `foreach` statement is a very convenient way of iterating through any enumerable collection type, from arrays to dictionaries.

You can see the difference in iterating collections using `for` loops or `foreach` by using the MeasureIt tool mentioned in Chapter 1. Using standard `for` loops is significantly faster in all the cases. However, if you do your own simple test, you might notice equivalent performance depending on the scenario. In some cases, .NET will actually convert simple `foreach` statements into standard `for` loops.

Take a look at the `ForEachVsFor` sample project, which has this code:

```
int[] arr = new int[100];
for (int i = 0; i < arr.Length; i++)
{
    arr[i] = i;
}

int sum = 0;
foreach (int val in arr)
{
    sum += val;
}

sum = 0;
IEnumerable<int> arrEnum = arr;
foreach (int val in arrEnum)
{
    sum += val;
}
```

Once you build this, then decompile it using an IL reflection tool. You will see that the first `foreach` is actually compiled as a `for` loop. The IL looks like this:

```
// loop start (head: IL_0034)
IL_0024: ldloc.s CS$6$0000
IL_0026: ldloc.s CS$7$0001
IL_0028: ldelem.i4
IL_0029: stloc.3
IL_002a: ldloc.2
IL_002b: ldloc.3
IL_002c: add
IL_002d: stloc.2
IL_002e: ldloc.s CS$7$0001
IL_0030: ldc.i4.1
IL_0031: add
IL_0032: stloc.s CS$7$0001
IL_0034: ldloc.s CS$7$0001
IL_0036: ldloc.s CS$6$0000
IL_0038: ldlen
IL_0039: conv.i4
IL_003a: blt.s IL_0024
// end loop
```

There are a lot of stores, loads, adds, and a branch – it is all quite simple. However, once we cast the array to an `IEnumerable<int>` and do the same thing, it gets a lot more expensive:

```

IL_0043: callvirt instance class
  [mscorlib]System.Collections.Generic.IEnumerator`1<!0>
    class [mscorlib]System.Collections.Generic.IEnumerable`1<int32>
      ::GetEnumerator()
IL_0048: stloc.s CS$5$0002
.try
{
  IL_004a: br.s IL_005a
  // loop start (head: IL_005a)
  IL_004c: ldloc.s CS$5$0002
  IL_004e: callvirt instance !0 class [mscorlib]
    System.Collections.Generic.IEnumerator`1<int32>
      ::get_Current()
  IL_0053: stloc.s val
  IL_0055: ldloc.2
  IL_0056: ldloc.s val
  IL_0058: add
  IL_0059: stloc.2

  IL_005a: ldloc.s CS$5$0002
  IL_005c: callvirt instance bool
    [mscorlib]System.Collections.IEnumerable::MoveNext()
  IL_0061: brtrue.s IL_004c
// end loop

  IL_0063: leave.s IL_0071
} // end .try
finally
{
  IL_0065: ldloc.s CS$5$0002
  IL_0067: brfalse.s IL_0070

  IL_0069: ldloc.s CS$5$0002
  IL_006b: callvirt instance void
    [mscorlib]System.IDisposable::Dispose()

  IL_0070: endfinally
} // end handler

```

We have 4 virtual method calls, a `try-finally`, and, not shown here, a memory allocation for the local enumerator variable which tracks the enumeration state. That is much more expensive than the simple `for` loop. It uses more CPU and more memory!

Remember, the underlying data structure is still an array (so a `for` loop is possible) but we are obfuscating that by casting to an `IEnumerable`. The important lesson here is the one that was mentioned at the top of the chapter: In-depth performance optimization will often defy code abstractions. `foreach` is an abstraction of a loop, and `IEnumerable` is an abstraction of a collection. Combined, they dictate behavior that defies the simple optimizations of a `for` loop over an array.

Casting

In general, you should avoid casting wherever possible. Casting often indicates poor class design, but there are times when it is required. It is relatively common to need to convert between unsigned and signed integers with different APIs, for example. Casting objects should be much rarer.

Casting objects is never free, but the costs differ dramatically depending on the relationship of the objects. Casting an object to its parent is relatively cheap. Casting a parent object to the correct child is significantly more expensive, and the costs increase with a larger hierarchy. Casting to an interface is more expensive than casting to a concrete type.

What you absolutely must avoid is an invalid cast. This will cause an exception of type `InvalidCastException` to be thrown, which will dwarf the cost of the actual cast by many orders of magnitude.

See the `CastingPerf` sample project in the accompanying source code which benchmarks a number of different types of casts. It produces this output on my computer in one test run:

```
No cast: 1.00x
Up cast (1 gen): 1.00x
```

```
Up cast (2 gens): 1.00x
Up cast (3 gens): 1.00x
Down cast (1 gen): 1.25x
Down cast (2 gens): 1.37x
Down cast (3 gens): 1.37x
Interface: 2.73x
Invalid Cast: 14934.51x
as (success): 1.01x
as (failure): 2.60x
is (success): 2.00x
is (failure): 1.98x
```

The `is` operator is a cast that tests the result and returns a Boolean value. The `as` operator is similar to a standard cast, but returns `null` if the cast fails. From the results above, you can see this is much faster than throwing an exception.

Never have this pattern, which performs two casts:

```
if (a is Foo)
{
    Foo f = (Foo)a;
}
```

Instead, use `as` to cast and cache the result, then test the return value:

```
Foo f = a as Foo;
if (f != null)
{
    ...
}
```

If you have to test against multiple types, then put the most common type first.

Note

One annoying cast that I see regularly is when using `MemoryStream.Length`, which is a `long`. Most APIs that use it are using the reference to the underlying buffer (retrieved from the `MemoryStream.GetBuffer` method), an offset, and a length, which is often an `int`, thus making a downcast from `long` necessary. Casts like these can be common and unavoidable.

Note that not all casting is explicit. You can have implicit casting that results in memory allocations, depending on how the classes are implemented.

P/Invoke

P/Invoke is used to make calls from managed code into unmanaged native methods. It involves some fixed overhead plus the cost of marshaling the arguments. Marshaling is the process of converting types from one format to another.

P/Invoke calls involve a bit of internal cleverness to make them work. A rough outline of the steps looks like this:

1. Adjust stack frame variables.
2. Set current stack frame.
3. Disable GC for the current thread.
4. Execute the target code.
5. Re-enable GC.
6. Check for a currently running GC and stop the thread if necessary.
7. Readjust stack frame variables back to their previous values.

You can see a simple benchmark of P/Invoke cost vs. a normal managed function call cost with the MeasureIt program mentioned in Chapter 1. On my computer, a P/Invoke call takes about 6–10 times the amount of time it takes to call an empty static method. You do not want to call a P/Invoked method in a tight loop if you have a managed equivalent, and you definitely want to avoid making multiple transitions between unmanaged and managed code. However, a single P/Invoke calls is not so expensive as to prohibit it in all cases.

There are a few ways to minimize the cost of making P/Invoke calls:

1. Avoid having a “chatty” interface. Make a single call that can work on a lot of data, where the time spent processing the data is significantly more than the fixed overhead of the P/Invoke call.
2. Use blittable types as much as possible. Recall from the discussion about structs that blittable types are those that have the same binary value in managed and unmanaged code, mostly numeric and pointer types. These are the most efficient arguments to pass because the marshaling process is essentially a memory copy.
3. Avoid calling ANSI versions of Windows APIs. For example, `CreateProcess` is actually a macro that resolves to one of two real functions, `CreateProcessA` for ANSI strings, and `CreateProcessW` for Unicode strings. Which version you get is determined by the compilation settings for the native code. You want to ensure that you are always calling the Unicode versions of APIs because all .NET strings are already Unicode, and having a mismatch here will cause an expensive, possibly lossy, conversion to occur.
4. Do not pin unnecessarily. Primitives are never pinned anyway and the marshaling layer will automatically pin strings and arrays of primitives. If you do need to pin something else, keep the object pinned for as short a duration as possible to. See Chapter 2 for a discussion of how pinning can negatively impact garbage collection. With pinning, you will have to balance this need for a short duration with the need to avoid a chatty interface. In all cases, you want the unmanaged code to return as fast as possible.

5. If you need to transfer a large amount of data to unmanaged code, consider pinning the buffer and having the native code operate on it directly. It does pin the buffer in memory, but if the function is fast enough this may be more efficient than a large copy operation. If you can ensure that the buffer is in gen 2 or the large object heap, then pinning is much less of an issue because the GC is unlikely to need to move the object anyway.
6. Decorate the imported method's parameters with the `In` and `Out` attributes. This will tell the CLR which direction each argument needs to be marshaled. For many types, this can be determined implicitly and you do not need to explicitly state it, such as for integer types. However, for strings and arrays, you should explicitly set this to avoid unnecessary marshaling in a direction you do not need.

Disable Security Checks for Trusted Code

For code you explicitly trust, you can reduce some of the cost of P/Invoke by disabling some security checks on the P/Invoke method declarations.

```
[DllImport("kernel32.dll", SetLastError=true)]
[System.Security.SuppressUnmanagedCodeSecurity]
static extern bool GetThreadTimes(IntPtr hThread,
                                  out long lpCreationTime,
                                  out long lpExitTime,
                                  out long lpKernelTime,
                                  out long lpUserTime);
```

The `SuppressUnmanagedCodeSecurity` attribute declares that the method can run with full trust. This will cause you to receive some Code Analysis (FxCop) warnings because it is disabling a large part of .NET's security model. You should disable this only if all of the following conditions are met:

1. Your application runs only trusted code.
2. You thoroughly sanitize the inputs, or otherwise run in a trusted environment.

3. You prevent public APIs from calling the P/Invoke methods

If you can do that, then you can gain some performance, as demonstrated in this MeasureIt output:

Name	Mean
PInvoke: 10 FullTrustCall() (10 call average) [count=1000 scale=10.0]	6.945
PInvoke: PartialTrustCall() (10 call average) [count=1000 scale=10.0]	17.778

The method running with full trust can execute about 2.5 times faster.

Delegates

There are two costs associated with use of delegates: construction and invocation. Invocation, thankfully, is comparable to a normal method call in nearly all circumstances. But delegates are objects and constructing them can be quite expensive. You want to pay this cost only once and cache the result. Consider the following code:

```
private delegate int MathOp(int x, int y);
private static int Add(int x, int y) { return x + y; }
private static int DoOperation(MathOp op, int x, int y)
{ return op(x, y); }
```

Which of the following loops is faster?

Option 1:

```
for (int i = 0; i < 10; i++)
{
    DoOperation(Add, 1, 2);
}
```

Option 2:

```
MathOp op = Add;
for (int i = 0; i < 10; i++)
{
    DoOperation(op, 1, 2);
}
```

It looks like Option 2 is only aliasing the `Add` function with a local `delegate` variable, but this actually causes a subtle change in memory allocation behavior! It becomes clear if you look at the IL for the respective loops:

Option 1:

```
// loop start (head: IL_0020)
IL_0004: ldnull
IL_0005: ldftn int32 DelegateConstruction.Program
    ::Add(int32, int32)
IL_000b: newobj instance void DelegateConstruction.Program/MathOp
    ::.ctor(object, native int)
IL_0010: ldc.i4.1
IL_0011: ldc.i4.2
IL_0012: call int32 DelegateConstruction.Program
    ::DoOperation(
        class DelegateConstruction.Program/MathOp,
        int32, int32)
...
...
```

While Option 2 has the same memory allocation, it is outside of the loop:

```
L_0025: ldnull
IL_0026: ldftn int32 DelegateConstruction.Program
    ::Add(int32, int32)
IL_002c: newobj instance void DelegateConstruction.Program/MathOp
    ::.ctor(object, native int)
...
// loop start (head: IL_0047)
IL_0036: ldloc.1
IL_0037: ldc.i4.1
IL_0038: ldc.i4.2
IL_0039: call int32 DelegateConstruction.Program
```

```

::DoOperation(class DelegateConstruction.Program/MathOp ,
    int32 , int32)
...

```

Notice the location of the newobj command has shifted up, above the loop start. The key to this issue is that delegates are backed by objects that are just like other objects. This goes for the built-in Func class as well. This means that if you want to avoid repeated allocation of delegate objects, you must reference them from a location that is called only once, as in the example above.

There is, however, a way of getting around this in an easy way: lambda expressions.

Consider what happens in this example:

```

for (int i = 0; i < 10; i++)
{
    DoOperation((x,y) => Add(x,y), 1, 2);
}

```

Here is the resulting IL code.

```

IL_004c: ldc.i4.0
IL_004d: stloc.3
IL_004e: br.s IL_007f
// loop start (head: IL_007f)
IL_0050: ldsfld class DelegateConstruction.Program/MathOp
    DelegateConstruction.Program/'<>c'::'<>9__3_0'
IL_0055: dup
IL_0056: brtrue.s IL_006f

IL_0058: pop
IL_0059: ldsfld class DelegateConstruction.Program/'<>c'
    DelegateConstruction.Program/'<>c'::'<>9'
IL_005e: ldftn instance int32
    DelegateConstruction.Program/'<>c'
    ::'<Main>b__3_0'(int32, int32)
IL_0064: newobj instance void
    DelegateConstruction.Program/MathOp
    ::.ctor(object, native int)
IL_0069: dup

```

```
IL_006a: stsfld class DelegateConstruction.Program/MathOp
          DelegateConstruction.Program/'<>c'::'<>9__3_0'

IL_006f: ldc.i4.1
IL_0070: ldc.i4.2
IL_0071: call int32 DelegateConstruction.Program
           ::DoOperation(class DelegateConstruction.Program/MathOp,
                         int32, int32)
...
// end loop
```

Notice that the delegate allocation is back inside the loop. However, look at line IL_0056 and you will see a `brtrue` instruction. This line is checking for the existence of a cached delegate. If it exists, then it will skip over the allocation directly to performing the operation. The loop still has extra instructions in it, but this is better than allocating on every loop iteration.

Note that the following syntax is equivalent to the previous example:

```
for (int i = 0; i < 10; i++)
{
    DoOperation((x,y) => { return Add(x, y); }, 1, 2);
}
```

These examples can be found in the DelegateConstruction sample project.

Exceptions

In .NET, putting a `try` block around code is cheap, but exceptions are very expensive to throw. This is largely because of the rich state that .NET exceptions contain, including doing a full stack walk. Exceptions must be reserved for truly exceptional situations, when raw performance ceases to be important.

Never rely on exception handling to catch simple error cases that would be more efficiently handled with non-exception code. It is much better to have validation code that can make simple checks and returns errors instead of throwing exceptions. This means that you must pay careful attention to your API design as you structure your program to efficiently handle errors.

To see the devastating effects on performance that throwing exceptions can have, see the ExceptionCost sample project. Its output should be similar to the following:

```
Empty Method: 1x
Exception (depth = 1): 8525.1x
Exception (depth = 2): 8889.1x
Exception (depth = 3): 8953.2x
Exception (depth = 4): 9261.9x
Exception (depth = 5): 11025.2x
Exception (depth = 6): 12732.8x
Exception (depth = 7): 10853.4x
Exception (depth = 8): 10337.8x
Exception (depth = 9): 11216.2x
Exception (depth = 10): 10983.8x
Exception (catchlist, depth = 1): 9021.9x
Exception (catchlist, depth = 2): 9475.9x
Exception (catchlist, depth = 3): 9406.7x
Exception (catchlist, depth = 4): 9680.5x
Exception (catchlist, depth = 5): 9884.9x
Exception (catchlist, depth = 6): 10114.6x
Exception (catchlist, depth = 7): 10530.2x
Exception (catchlist, depth = 8): 10557.0x
Exception (catchlist, depth = 9): 11444.0x
Exception (catchlist, depth = 10): 11256.9x
```

This demonstrates three simple facts:

1. A method that throws an exception is thousands of times slower than a simple empty method.
2. The deeper the stack for the thrown exception, the slower it gets (though it is already so slow, it does not matter).
3. Having multiple catch statements has a slight but perceptible effect as the right one needs to be found.

While catching exceptions may be cheap, accessing the `StackTrace` property on an `Exception` object can be very expensive as it reconstructs the stack from pointers and translates it into readable text. In a high-performance application, you may want to make logging of these stack traces optional through configuration and use it only when needed. Note that rethrowing an existing exception from an exception handler is the same expense as throwing a new exception.

To reiterate: exceptions should be truly exceptional. Using them as a matter of course can destroy your performance.

dynamic

It should probably go without saying, but to make it explicit: any code using the `dynamic` keyword, or the Dynamic Language Runtime (DLR) is not going to be highly optimized. Performance tuning is often about stripping away abstractions, but using the DLR is adding one huge abstraction layer. It has its place, certainly, but a fast system is not one of them.

When you use `dynamic`, what looks like straightforward code is anything but. Take a simple, admittedly contrived example:

```
static void Main(string[] args)
{
    int a = 13;
    int b = 14;

    int c = a + b;

    Console.WriteLine(c);
}
```

The IL for this is equally straightforward:

```
.method private hidebysig static
void Main (
    string[] args
) cil managed
```

```

{
// Method begins at RVA 0x2050
// Code size 17 (0x11)
.maxstack 2
.entrypoint
.locals init (
    [0] int32 a,
    [1] int32 b,
    [2] int32 c
)
IL_0000: ldc.i4.s 13
IL_0002: stloc.0
IL_0003: ldc.i4.s 14
IL_0005: stloc.1
IL_0006: ldloc.0
IL_0007: ldloc.1
IL_0008: add
IL_0009: stloc.2
IL_000a: ldloc.2
IL_000b: call void [mscorlib]System.Console::WriteLine(int32)
IL_0010: ret
} // end of method Program::Main

```

Now just make those ints dynamic:

```

static void Main(string[] args)
{
    dynamic a = 13;
    dynamic b = 14;

    dynamic c = a + b;

    Console.WriteLine(c);
}

```

For the sake of conserving print space, I will skip showing the IL here, but this is what it looks like when you convert it back to C#:

```

private static void Main(string[] args)
{

```

```
object a = 13;
object b = 14;
if (Program.<Main>o__SiteContainer0.<>p__Site1 == null)
{
    Program.<Main>o__SiteContainer0.<>p__Site1 =
        CallSite<Func<CallSite, object, object, object>>.
        Create(Binder.BinaryOperation(CSharpBinderFlags.None,
                                      ExpressionType.Add,
                                      typeof(Program),
                                      new CSharpArgumentInfo[]

        {
            CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                      null),
            CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                      null)
        }));
}
object c = Program.<Main>o__SiteContainer0.
    <>p__Site1.Target(Program.<Main>o__SiteContainer0.<>p__Site1,
                      a, b);
if (Program.<Main>o__SiteContainer0.<>p__Site2 == null)
{
    Program.<Main>o__SiteContainer0.<>p__Site2 =
        CallSite<Action<CallSite, Type, object>>.
        Create(Binder.InvokeMember(
            CSharpBinderFlags.ResultDiscarded,
            "WriteLine",
            null,
            typeof(Program),
            new CSharpArgumentInfo[]

        {
            CSharpArgumentInfo.Create(
                CSharpArgumentInfoFlags.UseCompileTimeType |
                CSharpArgumentInfoFlags.IsStaticType,
                null),
            CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None,
                                      null)
        }));
}
Program.<Main>o__SiteContainer0.<>p__Site2.Target(
    Program.<Main>o__SiteContainer0.<>p__Site2,
    typeof(Console), c);
}
```

Even the call to `WriteLine` is not straightforward. From simple, straightforward code, it has gone to a mishmash of memory allocations, delegates, dynamic method invocation, and these strange `CallSite` objects. A `CallSite` is how the DLR replaces standard method calls with a dynamically typed call. It wraps a sophisticated cache to avoid needing to do extensive reflection on every single method call. It is still expensive, however.

The JIT statistics are predictable:

Version	JIT Time	IL Size	Native Size
int	0.5ms	17 bytes	25 bytes
dynamic	10.9ms	209 bytes	389 bytes

I do not mean to dump too much on the DLR. It is a perfectly fine framework for rapid development and scripting. It opens up great possibilities for interfacing between dynamic languages and .NET, but it is not fast.

Reflection

Reflection is the process of programmatically iterating through loaded types and examining their metadata. It can also involve doing this to a dynamically loaded .NET assembly during runtime and executing methods on the found types. This is not a fast process under any circumstance. A .NET assembly's metadata is mostly organized for the purposes of loading, debugging, and offline tool access, not for runtime efficiency.

Getting information about all the types in an assembly is generally efficient it is just static metadata hanging around your process anyway. For example, here is some code that iterates through all types in the executing assembly and prints member method names:

```
foreach(var type in Assembly.GetExecutingAssembly().GetTypes())
{
    Console.WriteLine(type.Name);
    foreach(var method in type.GetMethods())
```

```
{  
    Console.WriteLine("\t" + method.Name);  
}  
}
```

It becomes less efficient as you start to dynamically allocate and execute code from that metadata. To demonstrate how reflection generally works in this scenario, here is some simple code from the ReflectionExe sample project that loads an “extension” assembly dynamically:

```
var assembly = Assembly.Load(extensionFile);  
  
var types = assembly.GetTypes();  
Type extensionType = null;  
foreach (var type in types)  
{  
    var interfaceType = type.GetInterface("IExtension");  
    if (interfaceType != null)  
    {  
        extensionType = type;  
        break;  
    }  
}  
  
object extensionObject = null;  
if (extensionType != null)  
{  
    extensionObject = Activator.CreateInstance(extensionType);  
}
```

At this point, there are two options we can follow to execute the code in our extension. To stay with pure reflection, we can retrieve the `MethodInfo` object for the method we want to execute and then invoke it:

```
MethodInfo executeMethod = extensionType.GetMethod("Execute");  
executeMethod.Invoke(extensionObject, new object[] { 1, 2 });
```

This is painfully slow, about 100 times slower than casting the object to an interface and executing it directly:

```
IExtension extensionViaInterface = extensionObject as IExtension;  
extensionViaInterface.Execute(1, 2);
```

If you can, you always want to execute your code this way rather than relying on the raw `MethodInfo.Invoke` technique. If a common interface is not possible, then see the next section on generating code to execute dynamically loaded assemblies much faster than reflection.

Code Generation

If you find yourself doing anything with dynamically loaded types (e.g., an extension or plugin model), then you need to carefully measure your performance when interacting with those types. Ideally, you can interact with those types via a common interface and avoid most of the issues with dynamically loaded code. This approach is described in Chapter 5 when discussing reflection. If that approach is not possible, use this section to get around the performance problems of invoking dynamically loaded code.

The .NET Framework supports dynamic type allocation and method invocation with the `Activator.CreateInstance` and `MethodInfo.Invoke` methods, respectively. Here is an example that uses both:

```
Assembly assembly = Assembly.Load("Extension.dll");  
Type type = assembly.GetType("DynamicLoadExtension.Extension");  
object instance = Activator.CreateInstance(type);  
  
MethodInfo MethodInfo = type.GetMethod("DoWork");  
bool result = (bool)MethodInfo.Invoke(instance, new object[]  
{ argument });
```

If you do this only occasionally, then it is not a big deal, but if you need to allocate a lot of dynamically loaded objects or invoke many dynamic function calls, these functions could become a severe bottleneck. `Activator.CreateInstance` not only uses significant CPU, but it can cause unnecessary allocations, which

put extra pressure on the garbage collector. There is also potential boxing that will occur if you use value types in either the function's parameters or return value (as the example above does).

If possible, try to hide these invocations behind an interface known both to the extension and the execution program, as described in the previous section. If that does not work, code generation may be an appropriate option. Thankfully, generating code to accomplish the same thing is quite easy.

Template Creation

To figure out what code to generate, use a template as an example to generate the IL for you to mimic. For an example, see the `DynamicLoadExtension` and `DynamicLoadExecutor` sample projects. `DynamicLoadExecutor` loads the extension dynamically and then executes `DoWork`. The `DynamicLoadExecutor` project ensures that `DynamicLoadExtension.dll` is in the right place with a post-build step and a solution build dependency configuration rather than project-level dependencies to ensure that code is indeed dynamically loaded and executed.

Start with creating a new extension object. To create a template, first understand what you need to accomplish. You need a method with no parameters that returns an instance of the type we need. Your program will not know about the `Extension` type, so it will just return it as an `object`. That method looks like this:

```
object CreateNewExtensionTemplate()
{
    return new DynamicLoadExtension.Extension();
}
```

Take a peek at the IL and it will look like this:

```
IL_0000: newobj instance void
[DynamicLoadExtension]DynamicLoadExtension.Extension
    ::.ctor()
IL_0005: ret
```

Delegate Creation

You can now create an instance of the `System.Reflection.Emit.DynamicMethod` type, programmatically add some IL instructions to it, and assign it to a delegate which you can then reuse to generate new `Extension` objects at will.

```
private static T GenerateNewObjDelegate<T>(Type type)
    where T:class
{
    // Create a new, parameterless (specified
    // by Type.EmptyTypes) dynamic method.
    var dynamicMethod = new DynamicMethod("Ctor_" + type.FullName,
                                           type,
                                           Type.EmptyTypes,
                                           true);
    var ilGenerator = dynamicMethod.GetILGenerator();

    // Look up the constructor info for the
    // type we want to create
    var ctorInfo = type.GetConstructor(Type.EmptyTypes);
    if (ctorInfo != null)
    {
        ilGenerator.Emit(OpCodes.Newobj, ctorInfo);
        ilGenerator.Emit(OpCodes.Ret);

        object del = dynamicMethod.CreateDelegate(typeof(T));
        return (T)del;
    }
    return null;
}
```

You will notice that the emitted IL corresponds exactly to our template method.

To use this, you need to load the extension assembly, retrieve the appropriate type, and pass it to the generator method.

```
Type type = assembly.GetType("DynamicLoadExtension.Extension");
Func<object> creationDel =
    GenerateNewObjDelegate<Func<object>>(type);
object extensionObj = creationDel();
```

Once the delegate is constructed you can cache it for reuse (perhaps keyed by the Type object, or whatever scheme is appropriate for your application).

Method Arguments

You can use the exact same trick to generate the call to the `DoWork` method. It is only a little more complicated due to a cast and the method arguments. IL is a stack-based language so arguments to functions must be pushed on to the stack in the correct order before a function call. The first argument for an instance method call must be the method's hidden `this` parameter that the object is operating on. Note that just because IL uses a stack exclusively, it does not have anything to do with how the JIT compiler will transform these function calls to assembly code, which often uses processor registers to hold function arguments.

As with object creation, first create a template method to use as a basis for the IL. Since we will have to call this method with just an `object` parameter (that is all we will have in our program), the function parameters specify the extension as just an `object`. This means we will have to cast it to the right type before calling `DoWork`. In the template, we have hard-coded type information, but in the generator we can get the type information programmatically.

```
static bool CallMethodTemplate(object extensionObj,
                               string argument)
{
    var extension = (DynamicLoadExtension.Extension)extensionObj;
    return extension.DoWork(argument);
}
```

The resulting IL for this template looks like:

```
.locals init (
    [0] class [DynamicLoadExtension] DynamicLoadExtension.Extension
        extension
)
IL_0000: ldarg.0
IL_0001: castclass
```

```
[DynamicLoadExtension] DynamicLoadExtension.Extension
IL_0006: stloc.0
IL_0007: ldloc.0
IL_0008: ldarg.1
IL_0009: callvirt instance bool
[DYNAMICLOADEXTENSION] DynamicLoadExtension.Extension
    ::DoWork(string)
IL_000e: ret
```

Notice that there is a local variable declared. This holds the result of the cast. We will see later that it can be optimized away. This IL leads to a straightforward translation into a `DynamicMethod`:

```
private static T GenerateMethodCallDelegate<T>(
    MethodInfo methodInfo,
    Type extensionType,
    Type returnType,
    Type[] parameterTypes) where T : class
{
    var dynamicMethod = new DynamicMethod(
        "Invoke_" + methodInfo.Name,
        returnType,
        parameterTypes,
        true);
    var ilGenerator = dynamicMethod.GetILGenerator();

    ilGenerator.DeclareLocal(extensionType);
    // object's this parameter
    ilGenerator.Emit(OpCodes.Ldarg_0);
    // cast it to the correct type
    ilGenerator.Emit(OpCodes.Castclass, extensionType);
    // actual method argument
    ilGenerator.Emit(OpCodes.Stloc_0);
    ilGenerator.Emit(OpCodes.Ldloc_0);
    ilGenerator.Emit(OpCodes.Ldarg_1);
    ilGenerator.EmitCall(OpCodes.Callvirt, methodInfo, null);
    ilGenerator.Emit(OpCodes.Ret);

    object del = dynamicMethod.CreateDelegate(typeof(T));
    return (T)del;
}
```

To generate the dynamic method, we need the `MethodInfo`, looked up from the extension's `Type` object. We also need the `Type` of the return object and the `Type` objects of all the parameters to the method, including the implicit `this` parameter (which is the same as `extensionType`).

To use our delegate, we just need to call it like this:

```
Func<object, string, bool> doWorkDel =
    GenerateMethodCallDelegate<
        Func<object, string, bool>>(
            MethodInfo, type, typeof(bool),
            new Type[]
                { typeof(object), typeof(string) });

bool result = doWorkDel(extension, argument);
```

Optimization

This method works perfectly, but look closely at what it is doing and recall the stack-based nature of IL instructions. Here is how this method works:

1. Declare local variable
2. Push `arg0` (the `this` pointer) onto the stack (`Ldarg_0`)
3. Cast `arg0` to the right type and push the result onto the stack (`Castclass`)
4. Pop the top of the stack and store it in the local variable (`Stloc_0`)
5. Push the local variable onto the stack (`Ldloc_0`)
6. Push `arg1` (the `string` argument) onto the stack (`Ldarg_1`)
7. Call the `DoWork` method (`Callvirt`)
8. Return

There is some glaring redundancy in there, specifically with the local variable. We have the casted object on the stack, we pop it off then push it right back on. We could optimize this IL by just removing everything having to do with the local variable. It is possible that the JIT compiler would optimize this away for us anyway, but doing the optimization does not hurt, and could help if we have hundreds or thousands dynamic methods, all of which will need to be JITted.

The other optimization is to recognize that the `callvirt` opcode can be changed to a simpler `call` opcode because we know there is no virtual method here. Now our IL looks like this:

```
var ilGenerator = dynamicMethod.GetILGenerator();

// object's this parameter
ilGenerator.Emit(OpCodes.Ldarg_0);
// cast it to the correct type
ilGenerator.Emit(OpCodes.Castclass, extensionType);
// actual method argument
ilGenerator.Emit(OpCodes.Ldarg_1);
ilGenerator.EmitCall(OpCodes.Call, methodInfo, null);
ilGenerator.Emit(OpCodes.Ret);
```

Wrapping Up

So how is performance with our generated code? Here is one test run:

```
==CREATE INSTANCE==
Direct ctor: 1.0x
Activator.CreateInstance: 14.6x
Codegen: 3.0x

==METHOD INVOKE==
Direct method: 1.0x
MethodInfo.Invoke: 17.5x
Codegen: 1.3x
```

Using direct method calls as a baseline, you can see that the reflection methods are much worse. Our generated code does not quite bring it back, but it is close. These numbers are for a function call that does not actually do anything, so they represent pure overhead of the function call, which is not a very realistic situation. If I add some minimal work (string parsing and a square root calculation), the numbers change a little:

```
==CREATE INSTANCE==  
Direct ctor: 1.0x  
Activator.CreateInstance: 9.3x  
Codegen: 2.0x
```

```
==METHOD INVOKE==  
Direct method: 1.0x  
MethodInfo.Invoke: 3.0x  
Codegen: 1.0x
```

In the end, this demonstrates that if you rely on `Activator.CreateInstance` or `MethodInfo.Invoke`, you can significantly benefit from some code generation.

STORY

I have worked on one project where these techniques reduced the CPU overhead of invoking dynamically loaded code from over 10% to something more like 0.1%.

You can use code generation for other things as well. If your application does a lot of string interpretation or has a state machine of any kind, this is a good candidate for code generation. .NET itself does this with regular expressions and XML serialization.

Preprocessing

If part of your application is doing something that is absolutely critical to performance, make sure it is not doing anything extraneous, or wasting time processing things that could be done beforehand. If data needs to be transformed before it is useful during runtime, make sure that as much of that transformation happens beforehand, even in an offline process if possible.

In other words, if something can be preprocessed, then it *must* be preprocessed. It can take some creativity and out-of-the-box thinking to figure out what processing can be moved offline, but the effort is often worth it. From a performance perspective, it is a form of 100% optimization by removing the code completely.

Investigating Performance Issues

Each of the topics in this chapter requires a different approach to performance. You can use the tools you already know from earlier chapters. CPU profiles will reveal expensive `Equals` methods, poor loop iteration, bad interop marshaling performance, and other inefficient areas.

Memory traces will show you boxing as object allocations and a general .NET event trace will show you where exceptions are being thrown, even if they are being caught and handled.

Performance Counters

The .NET CLR Interop category contains the following counters:

- # of CCWs: The number of COM-callable wrappers, or number of managed objects referred to by unmanaged COM objects.

- # of marshalling: Number of times arguments and return values have been marshaled by a P/Invoke stub. If the stub gets inlined (for very cheap calls), this value is not incremented. This is a good metric to track for how busy your calls to P/Invoke code are.
- # of Stubs: Number of stubs created by the JIT for marshaling arguments to P/Invoke or COM.

ETW Events

- **ExceptionThrown_V1:** An exception has been thrown. It does not matter if this exception is handled or not. Fields include:
 - Exception Type: Type of the exception.
 - Exception Message: Message property from the exception object.
 - EIPCodeThrow: Instruction pointer of throw site.
 - ExceptionHR: HRESULT of exception.
 - ExceptionFlags
 - 0x01: Has inner exception.
 - 0x02: Is nested exception.
 - 0x04: Is rethrown exception.
 - 0x08: Is a corrupted state exception.
 - 0x10: Is a CLS compliant exception.

Finding Boxing Instructions

It is fairly easy to scan your code for boxing because there is a specific IL instruction called `box`. To find it in a single method or class, just use one of the many IL decompilers available and select the IL view.

If you want to detect boxing in an entire assembly it is easier to use ILDASM, which ships with the Windows SDK, with its flexible command-line options.

This example analyzes `Boxing.exe` and outputs the IL code to `output.txt`

```
ildasm.exe /out=output.txt Boxing.exe
```

Take a look at the Boxing sample project, which demonstrates a few different ways boxing can occur. If you run ILDASM on Boxing.exe, you should see output similar to the following:

```
.method private hidebysig static void Main(string[] args)
    cil managed
{
    .entrypoint
// Code size      98 (0x62)
.maxstack 3
.locals init ([0] int32 val,
              [1] object boxedVal,
              [2] valuetype Boxing.Program/Foo foo,
              [3] class Boxing.Program/INameable nameable,
              [4] int32 result,
              [5] valuetype Boxing.Program/Foo '<>g__initLocal0')
IL_0000: ldc.i4.s    13
IL_0002: stloc.0
IL_0003: ldloc.0
IL_0004: box      [mscorlib]System.Int32
IL_0009: stloc.1
IL_000a: ldc.i4.s    14
IL_000c: stloc.0
IL_000d: ldstr     "val: {0}, boxedVal:{1}"
IL_0012: ldloc.0
IL_0013: box      [mscorlib]System.Int32
IL_0018: ldloc.1
IL_0019: call      string  [mscorlib]System.String::Format(string,
                                                               object,
                                                               object)
IL_001e: pop
IL_001f: ldstr     "Number of processes on machine: {0}"
IL_0024: call      class  [System]System.Diagnostics.Process[]
    [System]System.Diagnostics.Process::GetProcesses()
IL_0029: ldlen
IL_002a: conv.i4
IL_002b: box      [mscorlib]System.Int32
IL_0030: call      string  [mscorlib]System.String::Format(string,
                                                               object)
IL_0035: pop
```

```

IL_0036: ldloca.s    '<>g__initLocal0'
IL_0038: initobj   Boxing.Program/Foo
IL_003e: ldloca.s    '<>g__initLocal0'
IL_0040: ldstr      "Bar"
IL_0045: call       instance void Boxing.Program/Foo
                  ::set_Name(string)
IL_004a: ldloc.s    '<>g__initLocal0'
IL_004c: stloc.2
IL_004d: ldloc.2
IL_004e: box       Boxing.Program/Foo
IL_0053: stloc.3
IL_0054: ldloc.3
IL_0055: call       void Boxing.Program::UseItem(
                  class Boxing.Program/INameable)
IL_005a: ldloca.s    result
IL_005c: call       void Boxing.Program::GetIntByRef(int32&)
IL_0061: ret
} // end of method Program::Main

```

You can also discover boxing indirectly via PerfView. With a CPU trace, you can find excessive calling of the `JIT_New` function.

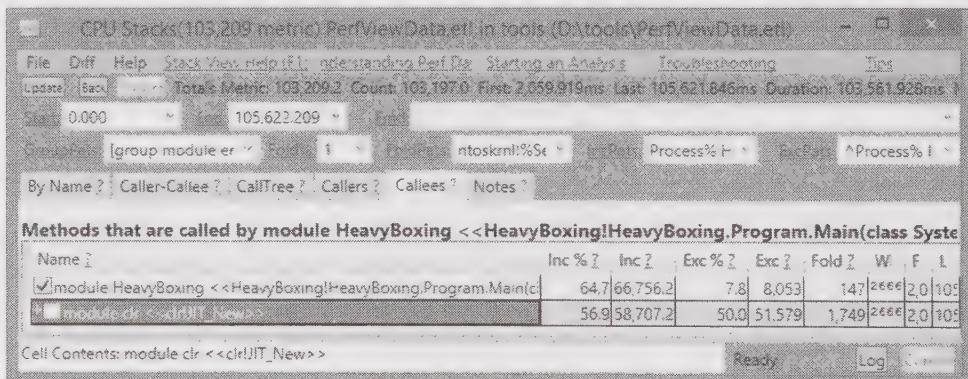


Figure 5.1. Boxing will show up in a CPU trace under the `JIT_New` method, which is the standard memory allocation method.

It is a little more obvious if you look at a memory allocation trace because you know that value types and primitives should not require a memory allocation at all.

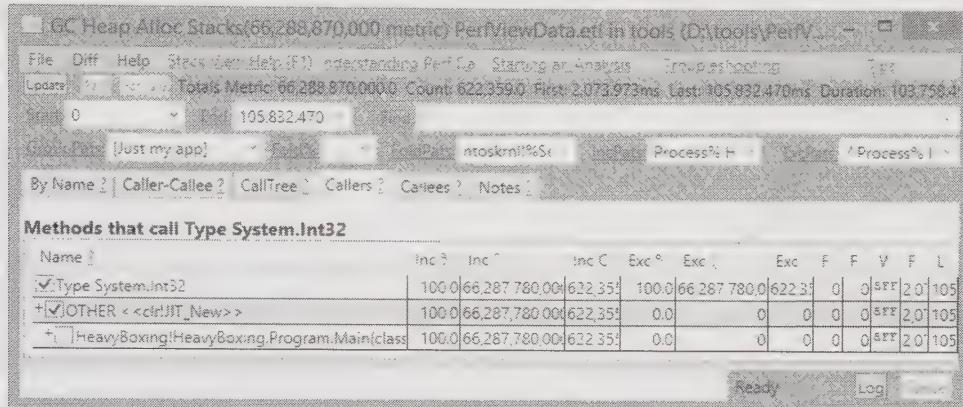


Figure 5.2. You can see in this trace that the *Int32* is being allocated via *new*, which should not feel right.

More directly, you can find any boxed object on the heap itself using CLR MD:

```
private static void PrintBoxedObjects(ClrRuntime clr)
{
    foreach (var obj in clr.Heap.EnumerateObjects())
    {
        if (obj.IsBoxed)
        {
            Console.WriteLine(
                $"0x{obj.Address:x} - {obj.Type.Name}");
        }
    }
}
```

Discovering First-Chance Exceptions

A first-chance exception is debugger-speak for an exception that is being surfaced before any possible exception-handlers have been discovered or called. A second-chance exception is one that is surfaced after handlers have been searched for in vain. A second-chance exception will likely crash the process.

WinDbg will break on second-chance exceptions by default, and you can control whether it breaks on first-chance exceptions with the `sx` command. To disable first-chance handling of CLR exceptions:

```
sxd clr
```

To re-enable them:

```
sxe clr
```

PerfView can easily show you which exceptions are being thrown, regardless of whether they are caught or not.

1. In PerfView, collect .NET events. The default settings are OK, but CPU is not necessary, so uncheck it if you need to profile for more than a few minutes.
2. When collection is complete, double-click on the “Exception Stacks” node.
3. Select the desired process from the list.
4. The Name view will show a list of the top exceptions. The CallTree view will show the stack for the currently selected exception.

Name	Inc %	Inc	Inc Ct
✓ ROOT	100.0	15,767.0	15,767
+✓ Process32 ExceptionCost.vshost (4640)	100.0	15,767.0	15,767
+✓ Thread (4828) CPU=0ms	100.0	15,767.0	15,767
+✓ OTHER <<ntdll!?>>	100.0	15,767.0	15,767
+✓ ExceptionCost!ExceptionCost.Program.Main(class System.)	100.0	15,767.0	15,767
+✓ ExceptionCost!ExceptionCost.Program.ExceptionMethod	100.0	15,767.0	15,767
+✓ ExceptionCost!ExceptionCost.Program.ExceptionMethod	87.3	13,766.0	13,766
+✓ OTHER <<clr!IL_Throw>>	87.3	13,766.0	13,766
+✓ ExceptionCost!ExceptionCost.Program.ExceptionMethod	87.3	13,766.0	13,766

Figure 5.3. PerfView makes finding where exceptions are coming from trivially easy.

Summary

Remember that in-depth performance optimizations will defy code abstractions. You need to understand how your code will be translated to IL, assembly code, and hardware operations. Take time to understand each of these layers.

Use a struct instead of a class when the data is relatively small, you want minimal overhead, or you are going to use them in arrays and want optimal memory locality. Consider making structs immutable and always implement `Equals`, `GetHashCode`, and `IEquatable<T>` on them. Avoid boxing of value types and primitives by guarding against assignment to object references.

Use `ref`-return for safe direct memory access to fields.

Keep iteration fast by not casting collections to `IEnumerable`. Avoid casting in general, whenever possible, especially instances that could result in an exception.

Minimize the number of P/Invoke calls by sending as much data per call as possible. Keep memory pinned as briefly as possible.

If you find yourself needing to make heavy use of `Activator.CreateInstance` or `MethodInfo.Invoke`, consider code generation instead.

Chapter 6

Using the .NET Framework

The previous chapter discussed general .NET coding techniques and pitfalls, especially those related to language features. In this chapter, we discuss some of the issues you must consider when using the enormous library of code that ships with .NET. I cannot possibly discuss all of the various subsystems and classes that are part of the .NET Framework, but the purpose of this chapter is to give you the tools you need to do your own investigations into performance, and to be aware of common patterns that you may need to avoid.

The .NET Framework was written with an extremely broad audience in mind (all developers everywhere, really), and is meant to be a general-purpose framework, providing stable, correct, robust code that can handle many situations. As such, it does not emphasize raw performance, and you will find many things you will need to work around in the inner loops of your codebase. The .NET Framework has to work for everybody, everywhere, making few assumptions about the calling code. It will often emphasize correctness and robustness over speed and efficiency. In your own code, however, you can often achieve large gains by understanding your own constraints and assumptions, and making appropriate customizations. This is not license to start reimplementing `string` on a new project, but to be aware of the limitations of the framework when combined with the critical areas of your own code's performance.

To get around weaknesses in the .NET Framework (or any 3rd-party library), you may need to use some ingenuity. Some possible approaches are:

- Use an alternate API with less cost.
- Redesign your application to not call the API as often.
- Re-implement some APIs in a more performant manner.
- Do an interop into a native API to accomplish the same thing (assuming the marshaling cost is less).

Understand Every API You Call

The guiding principle of this chapter is this:

You must understand the code executing behind every called API.

To say you have control of your performance is to assert that you know the code that executes in every critical path of your code. You should not have an opaque 3rd-party library at the center of your inner loop –that is ceding control.

You will not always have access to the source of every method you call (though you do always have access to the assembly level!), but there is usually good documentation for all Windows APIs. With .NET, you can use one of the many IL viewing tools out there to see what the Framework is doing. (This ease of inspection does not extend to the CLR itself, which, while it is available as part of .NET Core, is written largely in dense, heavily macroed native code.)

Get used to examining Framework code for anything you are not familiar with. The more that performance is important to you, the more you need to question the implementation of APIs you do not own. Just remember to keep your pickiness proportional to the need for speed.

What follows in this chapter is a discussion of a few general areas you should be concerned with as well as some specific, common classes every program will use.

Multiple APIs for the Same Thing

You will occasionally run into a situation where you can choose among many APIs for accomplishing the same thing. A good example is XML parsing. XML parsing is never fast, exactly, but there are better ways to do it depending on your scenario. There are at least 9 different ways to parse XML in .NET:

- `XmlTextReader`
- `XmlValidatingReader`
- `XDocument`
- `XmlDocument`
- `XPathNavigator`
- `XPathDocument`
- `LINQ-to-XML`
- `DataContractSerializer`
- `XmlSerializer`

Which one you use depends on factors such as ease of use, productivity, suitability for the task, and performance. `XmlTextReader` is very fast, but it is forward-only and does no validation. `XmlDocument` is very convenient because it has a full object model loaded, but it is among the slowest options.

It is as true for XML parsing as it for other API choices: not all options will be equal, performance-wise. Some will be faster, but use more memory. Some will use very little memory, but may not allow certain operations. You will have to determine which features you need and measure the performance to determine which API provides the right balance of functionality vs. performance. You should prototype the options and profile them running on sample data.

Collections

.NET provides over 21 built-in collection types, including concurrent and generic versions of many popular data structures. Most programs will only need to use a combination of these and you should rarely need to create your own.

Choosing which collections to use depends on many factors, including: semantic meaning in the APIs (push/pop, enqueue/dequeue, add/remove, etc.), underlying storage mechanism and cache locality, speed of various operations on the collection such as `Add` and `Remove`, and whether you need to synchronize access to the collection. All of these factors can greatly influence the performance of your program.

Collections to Avoid

Some collections still exist in the .NET Framework only for backward compatibility reasons and should never be used by new code. These include:

- `ArrayList`
- `Hashtable`
- `Queue`
- `SortedList`
- `Stack`
- `ListDictionary`
- `HybridDictionary`

The reasons these should be avoided are casting and boxing. These collections store references to `Object` instances in them so you will always need to cast down to your actual object type.

The boxing problem is even more pernicious. Suppose you want to have an `ArrayList` of `Int32` value types. Each value will be individually boxed and stored on the heap. Instead of iterating through a contiguous array of memory to access each integer, each array reference will require a pointer dereference, heap access (likely hurting locality), and then an unboxing operation to get at the inner value. This is horrible. Use a non-resizable array or one of the generic collection classes instead.

In the early versions of .NET there were some string-specific collections that are now obsolete because of the power of generics. Examples include `StringCollection`, `StringDictionary`, `NameValueCollection`, and `OrderedDictionary`. They do not necessarily have performance problems per se, but there is no need to even consider them unless you are using an existing API that requires them.

Arrays

The simplest, and likely the most-used, collection is the humble `Array`. Arrays are ideal because they are compact, using a single contiguous block of memory, which improves processor cache locality when accessing multiple elements (assuming value types – reference types will still jump to other places in the heap). Accessing them is in constant time and copying them is fast. Resizing them, however, will mean allocating a new array and copying the old values into the new object. Many of the more complicated data structures are built on top of arrays.

A common requirement is to be able to pass around a subsegment of an array. One way to do this is to copy the desired values into a new array of the correct size, but this incurs extra CPU and memory costs. A better way is to have a structure that just refers to the relevant portion of the array. Thankfully, there are two such structures already in .NET: `ArraySegment<T>` and `Span<T>`.

```
byte[] fileContents = File.ReadAllBytes("foo.bin");
ArraySegment<byte> header = new ArraySegment<byte>(fileContents,
    1,
    24);
ProcessHeader(header);
```

Many .NET APIs that can operate on arrays will often have a version that takes either an `ArraySegment<T>` directly, or the individual array reference, offset, and count values. When building your own APIs that operate on arrays, it would be wise to consider making an `ArraySegment<T>`-compatible version.

`Span<T>` is similar, but can represent sub-segments of multiple types of contiguous memory and was covered in detail in Chapter 2.

Jagged vs. Multi-dimensional Arrays

There are two ways to allocate multi-dimensional arrays in .NET:

- Multi-dimensional arrays: a single object with multiple indexes:

```
const int Size = 50;

private int[, ,] multiArray;

void Init()
{
    this.multiArray = new int[Size, Size, Size];
}
```

- Jagged arrays: arrays of arrays (multiple objects), each with a single index:

```
private const int Size = 50;

private int[][][] jaggedArray;

public void GlobalSetup()
{
    this.jaggedArray = new int[Size][][];
    for (int i = 0; i < Size; i++)
    {
```

```

        this.jaggedArray[i] = new int[Size]++;
        for (int j = 0; j < Size; j++)
        {
            this.jaggedArray[i][j] = new int[Size];
        }
    }
}

```

They look basically equivalent, but they are very different internally. The initialization code above does not really demonstrate this difference, so consider some code that iterates through all values and modifies each entry.

```

public void Negate_JaggedArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            for (int k = 0; k < Size; k++)
            {
                this.jaggedArray[i][j][k] *= -1;
            }
        }
    }
}

public void Negate_MultiArray()
{
    for (int i = 0; i < Size; i++)
    {
        for (int j = 0; j < Size; j++)
        {
            for (int k = 0; k < Size; k++)
            {
                this.multiArray[i, j, k] *= -1;
            }
        }
    }
}

```

The code looks extremely similar, but look at the IL code underneath, just for the inner loop. For the jagged array, it is pretty much what we expect:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[][][] ArrayPerf.ArrayPerfTest::jaggedArray
IL_0012: ldloc.0
IL_0013: ldelem.ref
IL_0014: ldloc.1
IL_0015: ldelem.ref
IL_0016: ldloc.2
IL_0017: ldelema [mscorlib]System.Int32
IL_001c: dup
IL_001d: ldind.i4
IL_001e: ldc.i4.m1
IL_001f: mul
IL_0020: stind.i4
IL_0021: ldloc.2
IL_0022: ldc.i4.1
IL_0023: add
IL_0024: stloc.2
```

It is just a series of memory accesses, some math, and stores. By contrast, here is the code for performing the same mutation on the multi-dimensional array:

```
IL_000c: ldarg.0
IL_000d: ldfld int32[0..., 0..., 0...]
    ArrayPerf.ArrayPerfTest::multiArray
IL_0012: ldloc.0
IL_0013: ldloc.1
IL_0014: ldloc.2
IL_0015: call instance int32& int32[0..., 0..., 0...>::
    Address(int32, int32, int32)
IL_001a: dup
IL_001b: ldind.i4
IL_001c: ldc.i4.m1
IL_001d: mul
IL_001e: stind.i4
IL_001f: ldloc.2
IL_0020: ldc.i4.1
IL_0021: add
IL_0022: stloc.2
```

It's mostly the same - except for that obvious method call in the middle. The difference this can make is demonstrated by the ArrayPerf project in the accompanying source:

Method	Mean	Error	StdDev
Negate_JaggedArray	281.0 us	1.7812 us	1.6661 us
Negate_MultiArray	439.6 us	0.2280 us	0.1780 us

The multi-dimensional arrays is significantly more expensive to traverse because of that method call. There should theoretically be some benefits in memory locality with multi-dimensional arrays and the ability to layout all values consecutively in memory, but it's not taken advantage of, and there is, frankly, very little reason to use multi-dimensional arrays in any circumstance.

Generic Collections

The generic collection classes are:

- `Dictionary< TKey , TValue >`
- `HashSet< T >`
- `LinkedList< T >`
- `List< T >`
- `Queue< T >`
- `SortedDictionary< TKey , TValue >`
- `SortedList< TKey , TValue >`
- `SortedSet< T >`
- `Stack< T >`

These deprecate all of the non-generic versions and should always be preferred. They incur no boxing or casting costs and will have better memory locality for value types (especially for the List-style structures that are implemented using arrays).

Within these collections, though, there can be very large performance differences. For example, `Dictionary`, `SortedDictionary`, and `SortedList` all store key-value relationships, but have very different insertion and lookup characteristics.

When discussing performance of collections (or more specifically, algorithms that operate on those collections), it is useful to have an abstract way of comparing them. In computer science, we use Big O notation for this. Briefly, Big O describes performance in terms of the problem size. For example, $O(n)$ means “linear time”—the time required is directly correlated with the problem size. A linear search through an unsorted list would be $O(n)$. $O(1)$ means “constant time”—the problem size is irrelevant, the operation always takes the same amount of time, as in hash table lookup. For more information about Big O notation, see Appendix C for a deeper discussion and more examples.

- `Dictionary` is implemented as a hash table and has $O(1)$ insertion and retrieval times.
- `SortedDictionary` is implemented as a binary search tree and has $O(\log n)$ insertion and retrieval times.
- `SortedList` is implemented as a sorted array. It has $O(\log n)$ retrieval times, but can have $O(n)$ insertion times in the worst case. If you insert random elements it will need to resize frequently and move the existing elements. It is ideal if you insert all of the elements in order, and then use it for fast lookups.

Of the three, `SortedList` has the smallest memory requirements. The other two will have much more random memory access, but can guarantee better insertion times on average. Which one of these you use depends greatly on your application’s requirements.

The difference between `HashSet` and `SortedSet` is similar to the difference between `Dictionary` and `SortedDictionary`.

- `HashSet` uses a hash table and has $O(1)$ insertion and removal operations.
- `SortedSet` uses a binary search tree and has $O(\log n)$ insertion and removal operations.

`List` has $O(1)$ insertion, but $O(n)$ removal and searching. `Stack` and `Queue` can only add or remove from one end of the collection so have $O(1)$ time in all operations.

`LinkedList` has $O(1)$ insertion and removal characteristics, but it should usually be avoided for large numbers of primitive types because it will allocate a new `LinkedListNode<T>` object for every item you add, which can be wasteful overhead.

To give you an idea of the difference in storage requirements for each collection discussed here, I ran a test program that added 1,000 4-byte integers into each structure and used `!ObjSize` in WinDbg to see the differences in used space:

- `List`: 4,036 bytes
- `Stack`: 4,036 bytes
- `Queue`: 4,044 bytes
- `SortedList`: 8,076 bytes
- `Dictionary`: 22,144 bytes
- `LinkedList`: 24,028 bytes
- `SortedSet`: 24,044 bytes
- `SortedDictionary`: 28,076 bytes
- `HashSet`: 30,972 bytes

You can see that `List`, `Stack`, and `Queue` are very close to the ideal storage requirements (4,000 bytes of pure data plus minimal overhead), but the others have varying amounts of overhead.

When choosing a data structure, first choose one that makes sense logically from the functionality point of view. If it gives good performance, then great. Otherwise, see if others have better insertion or retrieval characteristics for your scenario, and have acceptable storage requirements.

Concurrent Collections

The concurrent collection classes are safe to use across multiple threads without any additional thread synchronization. All of these classes are located in the `System.Collections.Concurrent` namespace and are all defined for use with generics:

- `ConcurrentBag<T>` (A bag is similar to a `Set`, but it allows duplicates)
- `ConcurrentDictionary< TKey , TValue >`
- `ConccurentQueue<T>`
- `ConcurrentStack<T>`

Most of these are implemented internally using `Interlocked` or `Monitor` synchronization primitives. Different methods require different levels of synchronization, so you must be careful. For example, calling `Count`, `IsEmpty`, `ToArray()`, and `TryUpdate()` on `ConcurrentDictionary` all involve taking a lock. You can and should examine their implementations using an IL reflection tool. Since every API is protected with some synchronization mechanism, frequent access may become expensive. As discussed in chapter 4, it may be better to forgo use of a concurrent collection and synchronize at a higher level.

Pay attention to the APIs for insertion and removal of items from these collections. They all have `Try-` methods which can fail to accomplish the operation in the case another thread beat them to it and there is now a conflict. For example, `ConcurrentStack` has a `TryPop` method which returns a `Boolean` value indicating whether it was able to pop a value. If another thread pops the last value, the current thread's `TryPop` will return `false`.

`ConcurrentDictionary` has a few methods which deserve special attention. You can call `TryAdd` to add a key and value to the dictionary, or `TryUpdate` to update an existing value. Often, you will not care whether it is already in the collection and want to add or update it – it does not matter. For this, there is the `AddOrUpdate` method which does exactly that, but rather than having you provide the new value directly, you instead need to pass two delegates: one for add and one for update. If the key does not exist, the first delegate will be called with the key and you will need to return a value. If the key does exist, the second delegate is called with the key and existing value and you need to return a new value (which could just be the existing value). You should pay attention to the issue of delegate caching, described in the previous chapter.

In either case, the `AddOrUpdate` method will return to you the new value – but it is important to realize that this new value may not be the value from the current thread's `AddOrUpdate` call! These methods are thread safe, but not atomic. It is possible another thread calls this method with the same key and the first thread will return the value from the second thread.

There is also an overload of the method that does not have a delegate for the add case (you just pass in a value).

A simple example will be helpful:

```
dict.AddOrUpdate(
    // Key I'm trying to add
    0,
    // Delegate to call when adding--
    // return string value based on the key
    key => key.ToString(),
    // Delegate to call when already present--update existing value
    (key, existingValue) => existingValue);

dict.AddOrUpdate(
    // Key I'm trying to add
    0,
    // Value to add if new
    "0",
    // Delegate to call when already present--update existing value
    (key, existingValue) => existingValue);
```

The reason for having these delegates rather than just passing in the value is that in many cases generating the value for a given key is a very expensive operation and you do not want two threads to do it simultaneously. The delegate gives you a chance to just use the existing value instead of regenerating a new copy. However, note that there is no guarantee that the delegates are called only once. Also, if you need to provide synchronization around the value creation or update, you need to add that synchronization in the delegates themselves - the collection will not do it for you.

Related to `AddOrUpdate` is the `GetOrAdd` method which has almost identical behavior.

```
string val1 = dict.GetOrAdd(
    // The key to retrieve
    0,
    // A delegate to generate the value if not present
    k => k.ToString());  
  
string val2 = dict.GetOrAdd(
    // The key to retrieve
    0,
    // The value to add if not present
    "0");
```

The lesson here is to be careful when using concurrent collections. They have special requirements and behaviors in order to guarantee safety and efficiency, and you need to understand exactly how they are used in the context of your program to use them correctly and effectively. Because of the semantics around delegates and the need to maintain a consistent data structure, you very well may execute more code than you think manipulating these data structures.

Bit-manipulation Collections

There are a handful of other specialized collections that ship with .NET, but most of them are string-specific or store `Object`, so can safely be ignored. Notable exceptions are `BitArray` and `BitVector32`.

`BitArray` represents an array of bit values. You can set individual bits and perform Boolean logic on the array as a whole. If you need only 32 bits of data, though, use `BitVector32` which is faster and has less overhead because it is a struct (it is little more than wrapper around an `Int32`).

Initial Capacity

Most collections use a simpler underlying data structure to store its data. Often, this is an array, or a set of arrays. As the collection fills up, these arrays may need to be re-allocated and the data copied. It can be easy to get into pathological cases where you spend most of your time just resizing data structures.

Thankfully, most collections susceptible to this problem provide a parameter in the constructor to pre-allocate a specific size. I recommend you use these parameters often, even in cases where you do not necessarily know the exact size you will end up with. As long as you do not extremely overestimate the capacity requirements, it will usually result in savings by avoiding repeated reallocations. In any case, you should measure this with a profiler.

Collection	Default Capacity	Can Pre-Allocate
<code>ConcurrentDictionary< TKey, TValue ></code>	31	yes
<code>Dictionary< TKey, TValue ></code>	0	yes
<code>HashSet< T ></code>	0	yes
<code>List< T ></code>	0	yes
<code>Queue< T ></code>	0	yes
<code>SortedDictionary< TKey, TValue >*</code>	0	no
<code>SortedList< T ></code>	0	yes
<code>SortedSet< T ></code>	0	no
<code>Stack< T ></code>	0	yes

`HashSet` will allocate prime-numbered space on the first insertion. `SortedSet` and `SortedDictionary` are implemented as a binary tree, which encapsulates individual nodes, allocated as needed.

Key Comparisons

Data structures which have a key component (e.g., `Dictionary<TKey, TValue>`) usually also take a comparer parameter, allowing you to decide what kind of key comparisons you want.

As you will see in the `String` section, doing a more appropriate (even if it is more expensive computationally) comparison is almost always preferable to manipulating the key for comparison's sake. The typical example is with strings and case-sensitivity.

I have actually seen code similar to the following because the code author did not know how to check keys correctly:

```
var keyToLookup = "myKey";
var dict = new Dictionary<string, object>();
...
foreach(var kvp in dict)
{
    if (kvp.Key.ToUpper() == keyToLookup.ToUpper())
    {
        ...
    }
}
```

This creates a tremendous amount of GC and memory waste, as well as unnecessary CPU cost. Beware of LINQ methods that make it easier to do these kinds of lookups as well.

There are a couple of ways to solve this:

1. Restrict the underlying dataset's keys to not require case-insensitive comparison. This will allow you to both avoid the memory allocation and get the cheapest comparison option.
2. Pass a comparison object to the `Dictionary<TKey, TValue>` constructor that can be used to perform more accurate comparisons and lookups. There are a number of comparers defined for you as static properties on the `StringComparer` class.

```
var keyToLookup = "myKey";
var dict = new Dictionary<string, object>(
    StringComparer.OrdinalIgnoreCase);
...
object val;
if (dict.TryGetValue(keyToLookup, out val))
{
    ...
}
```

Now the dictionary can be correctly used for its intended purpose.

Sorting

An additional consideration: If you ever use any of your own types (especially structs) as values in lists, and those values are inherently orderable, you should implement `IComparable<T>`.

```
struct MyType : IComparable<MyType>
{
    public string Name { get; set; }

    public int CompareTo(MyType other)
    {
        return this.Name.CompareTo(other.Name);
    }
}
```

If a value can be sorted in arbitrary ways (such as a `Person` object where sorting could be by birth date or by last name), then it is best to avoid implementing this altogether and avoid the potential confusion.

Creating Your Own Collection Types

I have rarely had the need to create my own collection types from scratch, but the need does occasionally arise. If the built-in types do not have the right semantics for you, then definitely create your own as an appropriate abstraction. When doing so, follow these general guidelines:

1. Implement the standard collection interfaces wherever they make sense
 - `IEnumerable<T>`
 - `ICollection<T>`
 - `IList<T>`
 - `IDictionary< TKey, TValue >`
2. Consider how the collection will be used when deciding how to store the data internally. Pay attention to things like locality-of-reference and favor arrays if sequential access is common.
3. Do you need to add synchronization into the collection itself? Or perhaps create a concurrent version of the collection?
4. Understand the run-time complexity of the add, insert, update, find, and remove algorithms. See Appendix C for a discussion of Big O complexity.
5. Implement APIs that make semantic sense, e.g., `Pop` for stacks, `Dequeue` for queues.

Strings

In .NET, strings have two problems:

1. Strings are immutable.
2. We want to mutate them.

From those two items spring most of the issues.

By immutable, we mean that, once created, strings exist forever in that state until garbage collected. This means that any modification of a string results in the creation of a new string. Fast, efficient programs generally do not modify strings in any way. Think about it: strings represent textual data, which is largely for human consumption. Unless your program is specifically for displaying or processing text, strings should be treated like opaque data blobs as much as possible. If you have the choice, always prefer non-string representations of data.

String Comparisons

As with so many things in performance optimization, the best string comparison is the one that does not happen at all. If you can get away with it, use enums, or some other numeric data for decision-making. If you must use strings, keep them short and use the simplest alphabet possible.

There are many ways to compare strings: by pure byte value, using the current culture, with case insensitivity, etc. You should use the simplest way possible. For example:

```
String.Compare(a, b, StringComparison.Ordinal);
```

is faster than

```
String.Compare(a, b, StringComparison.OrdinalIgnoreCase);
```

which is faster than

```
String.Compare(a, b, StringComparison.CurrentCulture);
```

If you are processing computer-generated strings, such as configuration settings or some other tightly coupled interface, then ordinal comparisons with case sensitivity are all you need.

All string comparisons should use method overloads that include an explicit `StringComparison` enumeration. Omitting this should be considered an error.

Finally, `String.Equals` is a special case of `String.Compare` and should be used when you do not care about sort order. It is not actually faster in many cases, but it conveys the intent of your code better.

ToUpper and ToLower

Avoid calling methods like `ToLower` and `ToUpper`, especially if you are doing this for string comparison purposes. Instead, use one of the `IgnoreCase` options for the `String.Compare` method.

There is a bit of a tradeoff, but not much of one. On the one hand, doing a case-sensitive string comparison is faster, but this still does not justify the use of `ToUpper` or `ToLower`, which are guaranteed to process every character, where a comparison operation can often make a decision as soon as the first non-equal character is hit—even faster if the strings differ in length. It also creates a new string, allocating memory and putting more pressure on the garbage collector. Just avoid this.

Concatenation

For simple concatenation of a known (at compile time) quantity of strings, you can usually just use the `+` operator or the `String.Concat` method. These are often more efficient in both CPU and memory usage than using a `StringBuilder`. There is also `String.Join`, but it uses `StringBuilder` internally.

```
string result = a + b + c + d + e + f;
```

Once the situation becomes more dynamic, it becomes trickier and can depend on factors such as the number of strings, their lengths, and whether you can pre-initialize the result buffer. At this point, consider using `StringBuilder`, which uses pooled char buffers to build up a string before you convert it to a `String` object.

The accompanying sample code contains the project StringConcatPerf, which benchmarks a number of concatenation methods.

As a baseline, it concatenates 10 literal strings, each of length 10. This takes about 0.001ns per iteration. With non-literals, the times break down as follows:

Count = 10, Length = 1

- `StringBuilder` (uninitialized): 99ns
- `StringBuilder` (initialized): 125ns
- `String.Concat`: 176ns

Count = 10, Length = 10

- `String.Concat`: 195ns
- `StringBuilder` (initialized): 246ns
- `StringBuilder` (uninitialized): 402ns

Count = 100, Length = 1

- `StringBuilder` (initialized): 771ns
- `StringBuilder` (uninitialized): 895ns
- `String.Concat`: 1714ns

Count = 100, Length = 10

- `StringBuilder` (initialized): 1721ns
- `String.Concat`: 1943ns

- `StringBuilder` (uninitialized): 2243ns

As you can see, the relative performance highly depends on the input characteristics. The difference between “initialized” `StringBuilder` and “uninitialized” `StringBuilder` is whether you pass an argument to the constructor to pre-allocate the needed amount of space:

```
var sb = new StringBuilder(1024); // pre-initialize
var sb2 = new StringBuilder();    // default 16-char buffer
```

If at all possible, you should pre-initialize `StringBuilder` capacity to be at least as large as what you may need to avoid repeated allocations of additional buffers. (`StringBuilder` does allocate additional buffers in chained chunks, so it does largely avoid the copying problem, but it is still an inefficient use of CPU and memory and can lead to more memory being allocated than you need.)

Formatting

`String.Format` is an expensive method. In addition to parsing the format string, it will box value type arguments, potentially call custom formatters, and allocate more memory than is strictly needed for the resulting string. Do not use it unless necessary. Avoid it for simple situations like this:

```
string message = String.Format(
    "The file {0} was {1} successfully.",
    filename,
    operation);
```

Instead, just do some simple concatenation:

```
string message = "The file " + filename + " was " + operation
+ " successfully";
```

Reserve use of `String.Format` for cases where performance does not matter or the format specification is more complex (like specifying how many decimals to use for a `double` value).

The `StringConcatPerf` sample project produces some benchmark numbers comparing `String.Format` vs `String.Concat`:

- `String.Concat`: 225ns
- `String.Format`: 440ns

It is nearly twice as expensive to produce the same string with `String.Format` compared to `String.Concat`. Use the simplest and cheapest tool for the job.

ToString

Be wary of calling `ToString` for many classes. If you are lucky, it will return a string that already exists. Other classes will cache the string once generated. For example, the `IPAddress` class caches its string, but it has an extremely expensive string generation process that involves `StringBuilder`, formatting, and boxing. Other types may create a new string every time you call it. This can be very wasteful for the CPU and can also impact the frequency of garbage collections.

When designing your own classes, consider the scenarios your class's `ToString` method will be called in. If it is called often, ensure that you are generating the string as rarely as possible. If it is only a debug helper, then it likely does not matter what it does.

`ToString` is often called by loggers and various string formatting helper methods like `String.Format` and `Console.WriteLine` (and related functionality in other classes). Make sure your own classes are designed with this in mind if possible. Some classes should produce a string only for debugging purposes.

Avoid String Parsing

If you can, design your system so that string parsing is not necessary. If data needs to be transformed from strings, can it be done offline or during startup? String processing is often CPU-intensive, repetitive, and memory-heavy – all things to avoid.

If you do need to parse during runtime, use methods that do not throw exceptions in failure circumstances. For example, do not use `Int32.Parse` because it will throw a `FormatException`. Instead, use `Int32.TryParse`, which will just return false. Many of the basic .NET types will have similar API options.

Substrings

The `String` class has various methods for returning portions of the string as new strings. These will allocate new objects. If you can possibly operate on a subportion in the form of a `char` array, consider using the `ReadOnlySpan<T>` struct as described in Chapter 2 to represent a portion of the underlying array, as in this example:

```
{  
    ...  
    ReadOnlySpan<char> subString =  
    "NonAllocatingSubstring".AsSpan().Slice(13);  
    PrintSpan(subString);  
    ...  
}  
  
private static void PrintSpan<T>(ReadOnlySpan<T> span)  
{  
    for (int i = 0; i < span.Length; i++)  
    {  
        T val = span[i];  
        Console.Write(val);  
        if (i < span.Length - 1) { Console.Write(", "); }  
    }  
    Console.WriteLine();  
}
```

The output of this code is:

S, u, b, s, t, r, i, n, g

Avoid APIs that Throw Exceptions Under Normal Circumstances

Exceptions are expensive, as you saw in Chapter 5. As such, they should be reserved for truly exceptional circumstances. Unfortunately, there are some common APIs that defy this basic assumption.

Most basic types have a `Parse` method, which will throw a `FormatException` when the input string is in an unrecognized format. For example, `Int32.Parse`, `DateTime.Parse`, etc. Avoid these methods in favor of `TryParse`, which will return a `bool` if parsing fails.

Another example is the `System.Net.HttpWebRequest` class, which will throw an exception if it receives a non-200 response from a server. This bizarre behavior is thankfully corrected in the `System.Net.Http.HttpClient` class in .NET 4.5.

Avoid APIs That Allocate From the Large Object Heap

Recall from chapter 2 that arrays are basically the only things that can be allocated large enough to be placed on the large object heap. If your own code is allocating these, they should of course be pooled.

Unfortunately, there are .NET APIs which will do this as well. The only way you can avoid these APIs is by profiling heap allocations using `PerfView`, which will show the stacks allocating memory like this. You cannot always avoid it, so just be aware that there are some .NET APIs that will do this. For example, calling the `Process.GetProcesses` method will guarantee an allocation on

the large object heap. You can avoid repeated LOH allocations by caching its results, calling it less frequently, or just avoid it completely by retrieving the information you need via interop directly from the Win32 API.

Some APIs will pool the large objects they create to avoid repeated allocation on the large object heap (`StringBuilder` does this, for example), but you would need to analyze their code to determine this. Profiling and going from there should be enough to steer you in the right direction.

Use Lazy Initialization

If your program uses a large or expensive-to-create object that is rarely used, or may not be used at all during a given invocation of the program, you can use the `Lazy<T>` class to wrap a lazy initializer around it. As soon as the `Value` property is accessed, the real object will be initialized according to the constructor you used to create the `Lazy<T>` object.

If your object has a default constructor, you can use the simplest version of `Lazy<T>`:

```
var lazyObject = new Lazy<MyExpensiveObject>();
...
if (needRealObject)
{
    MyExpensiveObject realObject = lazyObject.Value;
    ...
}
```

If construction is more complex, you can pass a `Func<T>` to the constructor.

```
var myObject = new Lazy<MyExpensiveObject>(
    () => Factory.CreateObject("A"));
...
MyExpensiveObject realObject = myObject.Value
```

`Factory.CreateObject` is a dummy method that produces an object of type `MyExpensiveObject`.

If `myObject.Value` is accessed from multiple threads, it is possible that each thread will need to initialize the object. By default, `Lazy<T>` is thread safe and only a single thread will be allowed to execute the creation delegate and set the `Value` property. You can modify this with a `LazyThreadSafetyMode` enumeration. This enumeration has three values:

- `None`: No thread safety. If important, you must ensure that the `Lazy<T>` object is accessed via a single thread in this case.
- `ExecutionAndPublication`: Only a single thread is allowed to execute the creation delegate and set the `Value` property. This is the default value if you use a constructor of `Lazy` without this option.
- `PublicationOnly`: Multiple threads can execute the creation delegate, but only a single one will initialize the `Value` property.

You should use `Lazy<T>` in place of your own singleton and double-checked locking pattern implementations.

If you have a large number of objects and `Lazy<T>` is too much overhead to use, you can use the static `EnsureInitialized` method on the `LazyInitializer` class. This uses `Interlocked` methods to ensure that the object reference is only assigned to once, but it does not ensure that the creation delegate is called only once. Unlike `Lazy<T>`, you must call the `EnsureInitialized` method yourself.

```
static MyObject[] objects = new MyObject[1024];

static void EnsureInitialized(int index)
{
    LazyInitializer.EnsureInitialized(ref objects[index],
        () => ExpensiveCreationMethod(index));
}
```

Note that by using the delegate here, you will have an extra allocation for the delegate object, in addition to whatever allocations `ExpensiveCreationMethod` performs.

The Surprisingly High Cost of Enums

You probably do not expect methods that operate on `Enum` values, a fundamentally integer type, to be very expensive. Unfortunately, because of the requirements of type safety, many simple operations are more expensive than you realize.

Take the `Enum.HasFlag` method, for example. You likely imagine the implementation to be something like the following:

```
public static bool HasFlag(Enum value, Enum flag)
{
    return (value & flag) != 0;
}
```

Unfortunately, what you actually get is something similar to:

```
// C# code generated by ILSpy
public bool HasFlag(Enum flag)
{
    if (flag == null)
    {
        throw new ArgumentNullException("flag");
    }
    if (!base.GetType().IsEquivalentTo(flag.GetType()))
    {
        throw new ArgumentException("Enum types do not match",
            new object[]
            {
                flag.GetType(),
                base.GetType()
            });
    }
    return this.InternalHasFlag(flag);
}
```

This is a good example of the side effects of using a general purpose framework. If you control your entire code base, then you can do better, performance-wise, by making certain assumptions and enforcing constraints not available to the .NET Framework. If you find you need to do a `HasFlag` test a lot, then do the check yourself:

```
[Flags]
enum Options
{
    Read = 0x01,
    Write = 0x02,
    Delete = 0x04
}

...
private static bool HasFlag(Options option, Options flag)
{
    return (option & flag) != 0;
}
```

`Enum.ToString` is also quite expensive for `Enums` that have the `[Flags]` attribute. One option to make it cheaper is to cache all of the `ToString` calls for that `Enum` type in a simple `Dictionary<EnumType, String>`. Or you can avoid writing these strings at all and get much better performance just using the actual numeric value and convert to strings offline.

For a fun exercise, see how much code is invoked when you call `Enum.IsDefined`. Again, the existing implementation is perfectly fine if raw performance does not matter, but you will be horrified if you find out it is a real bottleneck for you!

I found out about the performance problems of `Enum` the hard way, after a release. During a regular CPU profile I noticed that a significant portion of CPU, over 3%, was going to

just `Enum.HasFlag` and `Enum.ToString`. Excising all calls to `HasFlag` and using a `Dictionary` for the cached strings reduced the overhead to negligible amounts.

Tracking Time

Time means two things:

- Absolute time of day
- Time span (how long something took)

For absolute times, .NET supplies the versatile `DateTime` structure. However, calling `DateTime.Now` is a fairly expensive operation because it has to consider time zone information. Consider calling `DateTime.UtcNow` instead, which is more streamlined.

Even calling `DateTime.UtcNow` might be too expensive for you if you need to track a lot of time stamps. If that is the case, get the time once and then track offsets instead, rebuilding the absolute time offline, using the time span measuring techniques shown here.

To measure time intervals, .NET provides the `TimeSpan` struct. If you subtract two `DateTime` structs you will get a `TimeSpan` struct.

However, if you need to measure very small time spans with minimal overhead, use the system's performance counter, via `System.Diagnostics.Stopwatch` which will return to you a 64-bit number measuring the number of clock ticks since the CPU received power. To calculate the real time difference you take two measurements of the clock tick, subtract them, and divide by the system's clock tick count frequency. Note that this frequency is not necessarily related to the CPU's frequency. Most modern processors change their CPU frequency often, but the tick frequency will not be affected.

You can use the `Stopwatch` class like this:

```
var stopwatch = Stopwatch.StartNew();
//...do work...
stopwatch.Stop();
TimeSpan elapsed = stopwatch.Elapsed;
long elapsedTicks = stopwatch.ElapsedTicks;
```

There are also static methods to get a time stamp and the clock frequency, which may be more convenient if you are tracking a lot of time stamps and want to avoid the overhead of creating a new `Stopwatch` object for every interval.

```
long receiveTime = Stopwatch.GetTimestamp();
long parseTime = Stopwatch.GetTimestamp();
long startTime = Stopwatch.GetTimestamp();
long endTime = Stopwatch.GetTimestamp();

double totalTimeSeconds = (endTime - receiveTime) /
    Stopwatch.Frequency;
```

Finally, remember that values received from the `Stopwatch.GetTimestamp` method are only valid in the current executing session and only for calculating relative time differences.

Combining the two types of time, you can see how to calculate offsets from a base `DateTime` object to get new absolute times:

```
DateTime start = DateTime.Now;
long startTime = Stopwatch.GetTimestamp();
long endTime = Stopwatch.GetTimestamp();

double diffSeconds = (endTime - startTime) / Stopwatch.Frequency;
DateTime end = start.AddSeconds(diffSeconds);
```

Regular Expressions

A regular expression is a way to search for a string with advanced pattern-matching syntax.

Some simple examples include:

```
Regex regex = new Regex("<value>(.*)</value>");
```

This finds any text between some XML <value> tags. Regular expressions can be extremely complex and difficult to understand until you master their syntax and learn how to break them down into manageable chunks.

A more complex-looking expression is:

```
static Regex regex =
    new Regex("\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\\. [0-9]+)?)");
```

This extracts dollar values of the form “\$ 34.19”, “\$52”, and a few others.

Regular expressions are not fast. Internally, .NET creates a state machine to traverse the input string and match it against the pattern’s symbols. The costs include:

- Evaluation time can be long: This depends on the input text and the pattern to match. It is quite easy to write regular expressions that perform poorly. Minor differences in the expression can make significant differences in processing speed and optimizing them in and of themselves is a whole topic unto itself.
- Assembly generation: With some options, an in-memory assembly is generated on the fly when you create a `Regex` object. This helps with the runtime performance, but is expensive to create the first time.
- JIT costs can be high: The code generated from a regular expression can be very long and have patterns that give the JIT compiler fits. Thankfully, recent versions of the JIT have largely improved this.

As far as .NET goes, there are a few things you can do to improve the efficiency of `Regex` usage:

- Ensure you are up-to-date with .NET and patches. Specifically, .NET 4.6 has a new JIT compiler that dramatically improves regular expression parsing performance.
- Create a `Regex` instance variable rather than using the static methods. The static methods will internally create an instance anyway, but you will not be able to reuse it. For one-offs, these are fine.
- Create the `Regex` object with the `RegexOptions.Compiled` flag. Without this, regular expressions are interpreted. When compiled, expressions are much faster to evaluate, at the expense of steeper initialization costs due to emitting code into an in-memory assembly and then JITting it. You may also want to avoid compiling expressions if you are always creating new format strings and rarely reusing earlier ones. This will pollute the process with a lot of extra code.
- Do not recreate `Regex` objects over and over again. Create one, save it, and reuse it to match on new input strings. Never have a pattern like this:

```
class Foo
{
    private static void Evaluate(string[] inputs)
    {
        for (int i = 0; i < inputs.Length; i++)
        {
            Regex regex = new Regex("<value>(.*?)</value>",
                RegexOptions.Compiled);
            Match match = regex.Match(input);
            ...
        }
    }
}
```

Do this instead:

```
class Foo
{
    private static readonly Regex regex =
        new Regex("<value>(.*)</value>",
            RegexOptions.Compiled);

    private static void Evaluate(string[] inputs)
    {
        for (int i = 0; i < inputs.Length; i++)
        {
            Match match = regex.Match(inputs[i]);
            ...
        }
    }
}
```

You may also want to impose a timeout on complex expressions:

```
Regex regex = new Regex("<value>(.*)</value>",
    RegexOptions.Compiled,
    TimeSpan.FromSeconds(5));
```

You can also impose a global timeout via an application domain setting:

```
AppDomain.CurrentDomain.SetData("REGEX_DEFAULT_MATCH_TIMEOUT",
    TimeSpan.FromSeconds(10));
```

As an alternative to regular expressions, you can find or build your own parsing library for very simple patterns like dates or phone numbers that have few variations.

LINQ

The biggest danger with Language Integrated Query (LINQ) is that it has the potential to hide code from you – code for which you cannot be accountable because it is not present in your source file!

To illustrate this, look at a fairly simple example of a method that shifts all values in an array by a certain amount, provided they meet a certain condition:

```
public static int[] ShiftLinq(int[] vals, int shiftAmount)
{
    var temp = from n in vals select n + shiftAmount;
    return temp.ToArray();
}
```

It looks fairly innocuous, but there is a lot going on under the covers. Look at the IL this translates to:

```
.method public hidebysig static
    int32[] ShiftLinq (
        int32[] vals,
        int32 shiftAmount
    ) cil managed
{
    // Method begins at RVA 0x209c
    // Code size 37 (0x25)
    .maxstack 3
    .locals init (
        [0] class LinqCost.Program/'<>c__DisplayClass1_0'
    )

    IL_0000: newobj instance void LinqCost.Program
        '/'<>c__DisplayClass1_0'::ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: ldarg.1
    IL_0008: stfld int32
        LinqCost.Program/'<>c__DisplayClass1_0'
        ::shiftAmount
    IL_000d: ldarg.0
    IL_000e: ldloc.0
    IL_000f: ldftn instance int32
        LinqCost.Program/'<>c__DisplayClass1_0'
        ::'<ShiftLinq>b__0'(int32)
    IL_0015: newobj instance void class
        [mscorlib]System.Func`2<int32, int32>
        ::ctor(object, native int)
    IL_001a: call class [mscorlib]
```

```

System.Collections.Generic.IEnumerable`1<!!1>
[System.Core]System.Linq.Enumerable
::Select<int32, int32>(class
[mscorlib]System.Collections.Generic.IEnumerable`1
<!0>, class [mscorlib]System.Func`2<!0, !!1>)
IL_001f: call !!0[]
[System.Core]System.Linq.Enumerable
::ToArray<int32>(
    class [mscorlib]
        System.Collections.Generic.IEnumerable`1<!0>)
IL_0024: ret
} // end of method Program::ShiftLinq

```

You can see that there are 37 bytes of IL code, including two allocations for closure objects and delegates, as well as two method calls. Some of those method calls will have further allocations. To top it off, the final `ToArray` call at the end takes an `IEnumerable<T>` argument, which means that, depending on the source collection, it will not know how long the array is until it gets to the end. That implies continual resizing and copying until the end, then a final copy into a precisely sized array. Whether any of this matters to you depends on the needs of your program. It is certainly far more expensive than an alternative implementation:

```

public static void Shift(int[] vals, int shiftAmount)
{
    for (int i = 0; i < vals.Length; i++)
    {
        vals[i] += shiftAmount;
    }
}

```

Instead of query syntax, we have a typical `for`-loop with an `if` statement. This translates to the following IL:

```

.method public hidebysig static
    void Shift (
        int32[] vals,
        int32 shiftAmount
    ) cil managed
{

```

```

// Method begins at RVA 0x20d0
// Code size 27 (0x1b)
.maxstack 3
.locals init (
    [0] int32
)

IL_0000: ldc.i4.0
IL_0001: stloc.0
IL_0002: br.s IL_0014
// loop start (head: IL_0014)
    IL_0004: ldarg.0
    IL_0005: ldloc.0
    IL_0006: ldelema
        [mscorlib]System.Int32
    IL_000b: dup
    IL_000c: ldind.i4
    IL_000d: ldarg.1
    IL_000e: add
    IL_000f: stind.i4
    IL_0010: ldloc.0
    IL_0011: ldc.i4.1
    IL_0012: add
    IL_0013: stloc.0

    IL_0014: ldloc.0
    IL_0015: ldarg.0
    IL_0016: ldlen
    IL_0017: conv.i4
    IL_0018: blt.s IL_0004
// end loop

    IL_001a: ret
} // end of method Program::Shift

```

The resulting IL is smaller at 27 bytes, but more importantly there are no allocations or method invocations whatsoever.

If you are paying attention, you may notice a rather significant “cheat” in the for-loop version of the code: it modifies the original array, while the LINQ version creates a new one. This is deliberate. LINQ pushes you in the direction of creating new objects rather than reuse existing ones. This is often be-

cause LINQ makes such heavy use of the `IEnumerable<T>` interface, which is the lowest-level interface a collection can implement. This means that semantics more appropriate to the specific data structure cannot be used, and we lose some performance in the process. However, there are methods which will check for the presence of other interfaces like `ICollection` or `IList` and make use of those methods if possible. LINQ also takes a rather functional, immutable view of data. The lesson here is that your tools will often enforce design and performance constraints upon you.

So how does the performance actually differ? Here are the benchmark result numbers:

Method	Mean	Error	StdDev	Gen 0	Allocated
ShiftLinq	286.78 ns	1.8867 ns	1.7648 ns	0.0448	236 B
Shift	12.38 ns	0.1568 ns	0.1390 ns	-	0 B

The LINQ version is 20 times more expensive than the simpler version and because of its memory allocations, has a higher likelihood of causing garbage collections.

I want to be clear in stating that LINQ is not a bad technology—it has its place. It is phenomenally convenient at times, and many LINQ queries are perfectly performant, but it can make heavy use of delegates, interfaces, and temporary object allocation if you go crazy with temporary dynamic objects, joins, or complex `where` clauses.

You can often achieve some significant speedup in time by using Parallel LINQ, but keep in mind this is not actually reducing the amount of work to be done; it is just spreading it across multiple processors. For a mostly single-threaded application that just wants to reduce the time it takes to execute a LINQ query, this may be perfectly acceptable. On the other hand, if you are writing a server that is already using all of the cores to perform processing, then spreading LINQ across those same processors will not help the big picture and may even hurt it. In this case, it may be better to do without LINQ at all and find something more efficient.

If you suspect that you have some unaccounted-for complexity, run PerfView and look at the JITStats view to see the IL sizes and JIT times for methods that involve LINQ. Also look at the CPU usage of those methods once JITted.

It is worth noting that LINQ in .NET Core has made a number of efficiency improvements, such as reducing the number of allocations overall, avoiding delegate allocations, and improving algorithms all over the place to take advantage of known collection sizes (getting around the `IEnumerable` problem) that can offset some of the costs described here, but not all of them.

In the end, you must choose the appropriate tool for the job, and LINQ may be perfectly acceptable and convenient in most parts of your software, but inappropriate for other parts. Measure where allocations and CPU are going, and if it is in LINQ-heavy code, it is usually easy to convert it. Always let the profiler be the guide.

Reading and Writing Files

There are a number of convenience methods on the `File` class such as `Open`, `OpenRead`, `OpenText`, and `OpenWrite`. These are fine if performance is not critical.

If you are doing a lot of disk I/O, then you need to pay attention to the type of disk access you are doing, whether it is random, sequential, or if you need to ensure that the write has been physically written to the device before notifying the application of I/O completion. For this level of detail, you will need to use the `FileStream` class and a constructor overload that accepts the `FileOptions` enumeration. You can logically OR multiple flags together, but not all combinations are valid. None of these options are required, but they can provide hints to the operating system or file system on how to optimize file access.

```
using (var stream = new FileStream(
    @"C:\foo.txt",
    FileMode.Open,
    FileAccess.Read,
    FileShare.Read,
    16384 /* Buffer Size*/,
```

```
    FileMode.Create | FileMode.Truncate | FileMode.Open | FileMode.Append) &  
    FileOptions.SequentialScan | FileOptions.Encrypted))  
{  
    ...  
}
```

The options available to you are:

- **None**: No additional options.
- **Asynchronous**: Indicates that you will be doing asynchronous reading or writing to the file. This is not required to actually perform asynchronous reads and writes, but if you do not specify it, the underlying I/O will be performed synchronously without I/O completion ports. (Your thread will still execute asynchronously.) There are also overrides of the `FileStream` constructor that will take a Boolean parameter to specify asynchronous access.
- **DeleteOnClose**: Causes the OS to delete the file when the last handle to the file is closed. Use this for temporary files.
- **Encrypted**: Causes the file to be encrypted using the current account's credentials.
- **RandomAccess**: Gives a hint to the file system to optimize caching for random access.
- **SequentialScan**: Gives a hint to the file system that the file is going to be read sequentially from beginning to end.
- **WriteThrough**: Ignores caching and goes directly to the device's permanent storage. This generally makes I/O slower. The flag will be obeyed by the file system's cache, but many storage devices also have onboard caches, and they are free to ignore this flag and report a successful completion before it is written to permanent storage.

Random access is bad for any device, such as a hard disk or tape, that needs to seek to the required position. Sequential access should be preferred for performance reasons. However, many computers have SSDs these days where the difference between random and sequential access is minimal or nonexistent.

Most applications will not have to care about the specific style of I/O until they are completely saturating the devices they are accessing. When that occurs, you will need to take into account the type of hardware that exists, as well as what kind of accessing you are performing, and use the above flags accordingly.

Optimizing HTTP Settings and Network Communication

If your application makes outbound HTTP calls, there are a number of settings you can change to optimize network transmission. You should exercise caution in changing these, however, as their effectiveness greatly depends on your network topology and the servers on the other end of the connection. You also need to take into account whether the target endpoints are in a data center you control, or are somewhere on the Internet. You will need to measure carefully to see if these settings benefit you or not.

To change these by default for all endpoints, modify these static properties on the `ServicePointManager` class:

- `DefaultConnectionLimit`: The number of connections per end point. Setting this higher may increase overall throughput if the network links and both endpoints can handle it.
- `Expect100Continue`: When a client initiates a POST or PUT command it normally waits for a 100-Continue signal from the server before proceeding to send the data. This allows the server to reject the request before the data is sent, saving bandwidth. If you control both endpoints and this situation does not apply to you, turn this off to improve latency.
- `UseNagleAlgorithm`: Nagling, described in RFC 896 at <https://tools.ietf.org/html/rfc896.html> is a way to reduce the overhead of packets on a network by combining many small packets into a single larger packet. This can be beneficial by reducing overall network transmission overhead, but it can also cause packets to be delayed. On modern networks, this value should usually be off. You can experiment with turning this off and see if there is a reduction in response times.

Some of these settings can also be applied to individual `ServicePoint` objects, which can be useful if you want to customize settings by endpoint, perhaps to differentiate between local datacenter endpoints and those on the Internet. In addition to the above, the `ServicePoint` class also lets you control additional parameters:

- `ConnectionLeaseTimeout`: Specifies the maximum time in milliseconds that an active connection will be kept alive. Set this to `-1` to keep connections alive forever. This setting is useful for load balancing, where you will want to periodically force connections to close so that the process will reconnect to different machines. Setting this value to `0` will cause the connection to close after every request. This is not recommended because making a new HTTP connection is fairly expensive.
- `ConnectionLimit`: Specifies the maximum number of connections this endpoint can have.
- `MaxIdleTime`: Specifies the maximum time in milliseconds that a connection can remain open but idle. Set this to `Timeout.Infinite` to keep connections open indefinitely, regardless of whether they are active or not.
- `ReceiveBufferSize`: The size of the buffer used for receiving requests. The default is 8 KB. You can use a larger buffer if you regularly get large requests.
- `SupportsPipelining`: Allows multiple requests to be sent without waiting for a response between each one. However, the responses are sent back in order. See RFC 2616 at <https://tools.ietf.org/html/rfc2616> (the HTTP/1.1 standard) for more information.

You can also force an individual HTTP request to close its current connection (after the response has been sent back) by setting the `KeepAlive` header to `false`.

Ensure that what you are transmitting is optimally encoded. While profiling an internal system, we noticed an extremely high memory allocation rate and CPU usage for a particular component. With some investigation, we realized that it was receiving an HTTP response, transforming the received bytes into a base64-encoded string, decoding that string into a binary blob, and then finally deserializing that blob back into a strongly typed object. It was wasting bandwidth by needlessly encoding a binary blob as a string, and wasting our CPU with multiple layers of encoding, and finally it was causing more time spent in GC with multiple large object allocations. The lesson is to send only what you need, as compactly as possible. Base64 is rarely, if ever, useful today, especially among internal components. Regardless of whether you are doing file or network I/O, encode the data as ideally as possible. For example, if you need to read a series of integers, do not waste CPU, memory, disk space, and network bandwidth wrapping that in XML.

Finally, another word of caution relating to the principle highlighted at the top of this chapter about the general purpose of much of the .NET Framework. The built-in HTTP client, while generally very good and perfectly acceptable for downloading Internet content, may not be suitable for all applications, particularly if your application is very sensitive to latencies at high percentiles, and especially with intra-datacenter requests. If you care about 95th or 99th percentile latencies for HTTP requests, you may have to write your own HTTP client around the underlying WinHTTP APIs to get that last bit of performance. Doing this correctly takes quite a bit of expertise in both HTTP and multithreading in .NET to get right, so you need to justify the effort.

SIMD

Single-Instruction, Multiple-Data features of processors execute a single set of operations across multiple pieces of data. Most modern x64-platform processors from both Intel and AMD support these instructions, and they are critical for parallel computations required in things like gaming, scientific, and mathematical computation. The current JIT compiler as of this writing (4.7.1) makes some limited use of these instructions and registers, mostly through the System.Numerics.Vectors NuGet package.

For an example, consider an algorithm that finds the minimum value in an array. A straightforward serial algorithm would look like this:

```
int min = int.MaxValue;
for (int i = 0; i < sourceArray.Length; i++)
{
    if (sourceArray[i] < min)
    {
        min = sourceArray[i];
    }
}
```

The equivalent SIMD version of this is considerably more complex:

```
var minVector = new Vector<int>(int.MaxValue);
int i = 0;
// Process array in chunks of the vector's length
for (i=0; i < Length - Vector<int>.Count; i += Vector<int>.Count)
{
    Vector<int> subArray = new Vector<int>(this.sourceArray, i);
    minVector = Vector.Min(subArray, minVector);
}

// get min of the min vector and any leftover elements
int min = Int32.MaxValue;
for (int j = 0; j < Vector<int>.Count; j++)
{
    min = Math.Min(min, minVector[j]);
}
```

```

for (i = 0; i < sourceArray.Length; i++)
{
    min = Math.Min(min, sourceArray[i]);
}

```

Since `Vector` uses hardware registers, the length is governed by that hardware and is static, thus the use of the static property `Vector<int>.Count` in the example above. On my machine, the SIMD version is not any faster than the non-SIMD version. However, the situation is very different for another algorithm.

Here is an algorithm that scales an array by an integer value. The non-SIMD version is very simple:

```

for(int i=0; i < this.sourceArray.Length; i++)
{
    this.destinationArray[i] = this.sourceArray[i] * ScaleFactor;
}

```

The SIMD version is arguably simpler:

```
this.destinationVector = this.sourceVector * ScaleFactor;
```

In this case, the performance difference is enormous. Here are some benchmark results for both types of algorithms:

Method	Mean	Error	StdDev
Min_NonSIMD	4.312 ns	0.1615 ns	0.4761 ns
Min_SIMD	8.645 ns	0.0342 ns	0.0320 ns
Scale_NonSIMD	4.764 ns	0.0201 ns	0.0188 ns
Scale_SIMD	2.363 ns	0.0052 ns	0.0046 ns

You can see that scaling the vector is about twice as fast as the non-SIMD version.

There is no guarantee that any arbitrary SIMD-version of an algorithm is faster than its non-SIMD equivalent. Getting the algorithms right can depend on context, hardware, and the specifics of your CPU and memory patterns. There is

an overhead to using SIMD instructions and the trick is making this less than the speedup you get from the parallelism. Usually, the effectiveness of a SIMD-based algorithm will depend on how effectively you can keep the algorithm operating on data within the `Vector` struct or other `System.Numerics.Vector` data structures. This is the equivalent of keeping the calculations on the hardware. When you transfer values from the `Vector` to standard .NET data structures, you lose the benefits of the parallelism and hardware acceleration.

Investigating Performance Issues

Many of the techniques for finding issues with .NET Framework performance are exactly the same as with your own code. When you use tools to profile CPU usage, memory allocations, exceptions, contention, and more, you will see the hot spots in the framework just like you see them in your own code.

Note that PerfView will group much of the framework together and you may need to change the view settings to get a better picture of where Framework performance is going.

Performance Counters

.NET has many categories of performance counters. Chapters 2 through 4, which cover garbage collection, JIT compilation, and asynchronous programming, all detail the performance counters for their specific topic. .NET has additional performance counters for the following categories:

- **.NET CLR Data:** Counters relating to SQL clients, connection pools, and commands.
- **.NET CLR Exceptions:** Counters relating to rate of exceptions thrown.
- **.NET CLR Interop:** Counters relating to calling native code from managed code.

- **.NET CLR Networking:** Counters relating to connections and amount of data transmitted.
- **.NET CLR Remoting:** Counters relating to the number of remote calls, object allocations, channels, and more.
- **.NET CLR Data Provider for SqlServer/Oracle:** Counters for various .NET database clients.

Depending on your system's configuration you may see more or less than these.

Summary

As with all frameworks, you need to understand the implementation details of all the APIs you use. Do not take anything for granted.

Take care when picking collection classes. Consider API semantics, memory locality, algorithmic complexity, and space usage when choosing a collection. Prefer jagged arrays over multi-dimensional arrays. Completely avoid the older-style non-generic collections like `ArrayList` and `HashTable`. Use concurrent collections only when you need to synchronize most or all of the accesses. Consider initial capacity, key comparisons, and sorting ability with collections and the objects they store.

Pay particular attention to string usage and avoid creating extra strings. Prefer simpler APIs over more complex ones like `String.Format`. Use `ReadOnlySpan<T>` for substrings, when possible.

Avoid APIs that throw exceptions in normal circumstances, allocate from the large object heap, or have more expensive implementations than you expect.

When using regular expressions, make sure that you do not recreate the same `Regex` objects over and over again, and strongly consider compiling them with the `RegexOptions.Compiled` flag.

Pay attention to the type of I/O you are doing and use the appropriate flags when opening files to give the OS a chance to optimize performance for you. For network calls, disable Nagling and `Expect100Continue`. Only transmit the data you need and avoid unnecessary layers of encoding.

Avoid using reflection APIs to execute dynamically loaded code. Call this kind of code via common interfaces or through code-generated delegates.

If you are doing significant manipulation of arrays and matrices, use the `System.Numerics.Vectors` NuGet package to take advantage of SIMD commands.

Chapter 7

Performance Counters

Performance counters are critical for tracking the overall performance of your application over time. If you are responsible for tracking and improving the performance of your system, performance counters will help you do that. While you can (and should) use them for real-time monitoring, they can be even more valuable if you store them in a database for analysis over long periods of time. In this way, you can see how new releases, usage patterns, or other events affect the performance of your application.

You can consume performance counters in your own code for self-monitoring, archiving, or automated analysis. You can also create and register your own counters which are then available to the same monitoring approach. By correlating your program's custom counters with the system's counters for your application, you can often find the source of problems very quickly.

Performance counters are Windows-managed objects that track values over time. These values can be arbitrary numbers, counts, rates, time spans, or other kinds. Each counter has a category and a name associated with it. Most counters also have instances, which are specific subdivisions by logical, discrete entities. For example, the `% Processor Time` counter in the `Processor` category has instances for each process currently running. Many counters also have meta-instances like `Total` or `<Global>` that aggregate the data across all instances. Many components in Windows create their own performance counters and the CLR is no exception. There are hundreds of counters available to you to track

nearly every aspect of your program's performance. These counters are all described in the relevant chapters earlier in this book. To see all the installed performance counters on a system, use the PerfMon.exe utility that comes with Windows, as described in Chapter 1. The current chapter discusses programmatic access to these counters, both consuming and creating your own.

Consuming Existing Counters

To consume a counter, you need to create an instance of the `PerformanceCounter` class, supplying it with the category and name you want to monitor. You can optionally specify the instance and machine name as well. Here is an example that attaches the counter object to the % Processor Time counter:

```
PerformanceCounter cpuCtr = new PerformanceCounter("Process",
    "% Processor Time", process.ProcessName);
```

To retrieve values, you periodically call the `NextValue` method on the counter:

```
float value = cpuCtr.NextValue();
```

The API documentation recommends that you call `NextValue` no more frequently than once per second to allow the counter sufficient time to do the next read. To see a simple project that demonstrates consuming multiple built-in and custom counters, see the accompanying `PerfCountersTypingSpeed` sample.

Creating a Custom Counter

To create your own custom counter, create an instance of the `CounterCreationData` class, supplying a name and type. You add this to a collection, which is then added to a category.

```
const string CategoryName = "PerfCountersTypingSpeed";
```

```
if (!PerformanceCounterCategory.Exists(CategoryName))
{
    var counterDataCollection = new CounterCreationDataCollection();

    var wpmCounter = new CounterCreationData();
    wpmCounter.CounterType =
        PerformanceCounterType.RateOfCountsPerSecond32;
    wpmCounter.CounterName = "WPM";
    counterDataCollection.Add(wpmCounter);

    try
    {
        // Create the category.
        PerformanceCounterCategory.Create(
            CategoryName,
            "Demo category to show how to create and consume counters",
            PerformanceCounterCategoryType.SingleInstance,
            counterDataCollection);
    }
    catch (SecurityException)
    {
        // Handle error -- no permissions to make this change!
    }
}
```

To create a custom counter, you must have `PerformanceCounterPermission` granted. In practice, this means that you should usually create new counters with an installation program that can run with elevated permissions. .NET provides the `PerformanceCounterInstaller` class, which can wrap multiple instances of the `CounterCreationData` class and install them for you, with support for rollback and removal.

There are many types of counters, grouped into a few categories, as detailed next. In addition, some counters have 32-bit and 64-bit sizes defined. You can use whichever one is most appropriate for the data you are recording.

Averages

These counters show the average of the last two measurements.

- **AverageCount64**: How many things are processed during an operation.
- **AverageTimer32**: How long it takes to process an operation.
- **CountPerTimeInterval32/64**: The average length of a queue for a resource.
- **SampleCounter**: Counts the number of operations completed in a second.

The **AverageCount64** and **AverageTimer32** counters need the help of a second counter, **AverageBase**, to determine how many operations were completed since the last time the counter was updated. The **AverageBase** counter must be initialized right after the counter to which it applies. The following code demonstrates how to create these two counters together:

```
var counterDataCollection = new CounterCreationDataCollection();

// Actual average counter
var bytesPerTx = new CounterCreationData();
bytesPerTx.CounterType = PerformanceCounterType.AverageCount64;
bytesPerTx.CounterName = "BytesPerTransmission";
counterDataCollection.Add(bytesPerTx);

// Base counter to help in calculations
var bytesPerTxBase = new CounterCreationData();
bytesPerTxBase.CounterType = PerformanceCounterType.AverageBase;
bytesPerTxBase.CounterName = "BytesPerTransmissionBase";
counterDataCollection.Add(bytesPerTxBase);

PerformanceCounterCategory.Create(
    "Network Statistics",
    "Network statistics demo counters",
    PerformanceCounterCategoryType.SingleInstance,
    counterDataCollection);
```

To set the values, you adjust each counter according to the number of items and the number of operations to which those items apply. In this example, it is fairly simple:

```
bytesPerTx.IncrementBy(request.Length);  
bytesPerTxBase.IncrementBy(1);
```

Instantaneous

These are the simplest counters. They just reflect the most recent sample value.

- **NumberOfItems32/64:** Most recent value for a raw number.
- **NumberOfItemsHEX32/64:** Same as **NumberOfItems32/64**, displayed in hexadecimal format.
- **RawFraction:** With **RawBase** base counter, shows a percentage of the total. The total value is given to the base counter and the subset of that total value is assigned to this counter. An example would be percentage of a disk in use.

Deltas

Delta counters show the difference between the last two counter values.

- **CounterDelta32/64:** Shows the difference between the last two recorded values.
- **Elapsed Time:** Shows the time from when the component or process was started to now. For example, you can use this to track your application's up-time. You do not provide new values to this counter after initialization.
- **RateOfCountsPerSecond32/64:** Average number of operations completed per second.

Percentages

Percentage counters show the percent of a resource being used. In some cases, this percent can be greater than 100%. Consider a multiprocessor system: you could represent the CPU usage as a percentage of a single core. Each instance of the counter represents one of the cores. If multiple cores are simultaneously in use, then the percentage will be larger than 100.

- **CounterTimer**: Percent time a component is active in the total sample time.
- **CounterTimerInverse**: Similar to **CounterTimer**, except that it is measuring the time a component is NOT active and then subtracting that from 100%. In other words, this has the same value as **CounterTimer**, but arrives at the value in an inverse way.
- **CounterMultiTimer**: Similar to **CounterTimer**, but the active time is aggregated over all the instances, which can lead to percentages larger than 100.
- **CounterMultiTimerInverse**: Multiple instances, but derived from inactive time.
- **CounterMultiTimer100Ns**: Uses 100 nanosecond ticks instead of the system's performance counter ticks.
- **CounterMultiTimer100NsInverse**: Similar to **CounterMultiTimer100Ns**, but using the inverse logic.
- **SampleFraction**: The ratio of a subset of values to the total number of values. Uses the **SampleBase** base counter to track the total number of values.
- **Timer100Ns**: Percent time a component is active in the total sample time, measured in 100ns increments.
- **Timer100NsInverse**: Same as **Timer100Ns**, but uses the inverse logic.

All of the CounterMulti- counters require the use of the CounterMultiBase, similar to the AverageCount64 example earlier.

When creating your own performance counters, note that you should not update them too often. A maximum of once per second is a good rule because the data will never be exposed more frequently than that. If you need to generate high-volume performance data, ETW events are a much better choice.

Summary

Performance counters are the most basic building block of performance analysis. While your program does not necessarily have to create its own, consider doing so if you have discrete operations, phases, or items that have performance impact.

Consider automated ingestion and analysis of counters via .NET APIs to provide archiving and continual feedback about system performance.

Chapter 8

ETW Events

The previous chapter discussed performance counters which are excellent for tracking your application's overall performance. What performance counters cannot do is provide any information on a specific event or operation in your application. For that, you need to log data per operation. If you include time stamps then you now have the ability to track performance in your program in an extremely detailed way.

There are many logging libraries for .NET. There are some popular ones like log4net, as well as countless custom solutions. However, I strongly encourage you to use Event Tracing for Windows, which has a number of advantages:

1. It is built into the operating system.
2. It is extremely fast and suitable for high-performance scenarios. (However, it is not without cost, especially in some scenarios. It is possible to max it out with a large, multi-threaded application that writes a lot of events; resource-intensive games may also need a custom event system.)
3. It has automatic buffering.
4. You can dynamically enable and disable consuming and producing events during runtime.

5. It has a highly selective filtering mechanism.
6. You can integrate events from multiple sources into one log stream for comprehensive analysis.
7. All OS and .NET subsystems generate ETW events.
8. Events are typed and ordinal-based instead of just strings.

PerfView and many other profiling tools are nothing more than fancy ETW analyzers. For example, in Chapter 2 you saw how to use PerfView to analyze memory allocations. All of this information came from ETW events from the CLR.

In this chapter, you will explore how to define your own events and then consume them. All of the classes are in the `System.Diagnostics.Tracing` namespace and are available as of .NET 4.5.

You can define events that mark the start and stop of your program, the various stages of processing requests, errors that occur, or literally anything else. You have complete control of what information goes into an event.

Using ETW starts by defining things called providers. In .NET terms, this is a class with methods that define the events you want to log. These methods can accept any fundamental data type in .NET, such as strings, integers, and more.

Events are consumed by objects called listeners, which subscribe to the events they are interested in. If there are no subscribers for an event, then those events are discarded. This makes ETW very cheap on average.

Defining Events

Events are defined with a class that derives from the `EventSource` class, as in this example:

```
using System.Diagnostics.Tracing;

namespace EtlDemo
{
    [EventSource(Name="EtlDemo")]
    internal sealed class Events : EventSource
    {
        ...
    }
}
```

The `Name` argument is necessary if you want listeners to find the source by name. You can also provide a GUID, but this is optional and if you do not provide one, it will be generated automatically for you from the name by a procedure specified in RFC 4122 (See <https://tools.ietf.org/html/rfc4122>). GUIDs are only necessary if you need to guarantee an unambiguous event source. If your source and listener are all in one process then you do not even need a name, and can pass the `EventSource`-derived object directly to the listener.

After this basic definition, there are some conventions you should follow in defining your events. To illustrate them, I will define some events for a very simple test program (see the `EtlDemo` sample project).

```
using System.Diagnostics.Tracing;

namespace EtlDemo
{
    [EventSource(Name="EtlDemo")]
    internal sealed class Events : EventSource
    {
        public static readonly Events Log = new Events();

        public class Keywords
        {
            public const EventKeywords General = (EventKeywords)1;
            public const EventKeywords PrimeOutput = (EventKeywords)2;
        }

        internal const int ProcessingStartId = 1;
        internal const int ProcessingFinishId = 2;
    }
}
```

```
internal const int FoundPrimeId = 3;

[Event(ProcessingStartId,
       Level = EventLevel.Informational,
       Keywords = Keywords.General)]
public void ProcessingStart()
{
    if (this.IsEnabled())
    {
        this.WriteEvent(ProcessingStartId);
    }
}

[Event(ProcessingFinishId,
       Level = EventLevel.Informational,
       Keywords = Keywords.General)]
public void ProcessingFinish()
{
    if (this.IsEnabled())
    {
        this.WriteEvent(ProcessingFinishId);
    }
}

[Event(FoundPrimeId,
       Level = EventLevel.Informational,
       Keywords = Keywords.PrimeOutput)]
public void FoundPrime(long primeNumber)
{
    if (this.IsEnabled())
    {
        this.WriteEvent(FoundPrimeId, primeNumber);
    }
}
```

The first thing declared is a static reference to an instance of itself, named Log. This is common because events are usually global in nature and need to be accessed by many parts of your code. Having this “global” variable makes this a

lot easier than passing a reference to the `EventSource`-derived object around to everything that needs it. You can name it whatever you want, but you should standardize on the same term for all your event sources for clarity.

After the declaration, there is an inner class which just defines some constant **Keyword** values. **Keywords** are optional and their values are arbitrary and completely up to you. They serve as a way of categorizing events in ways that are meaningful to your application. Listeners can filter what they want to listen for based on **Keywords**. Note that **Keywords** are treated like bit flags so you must assign values of powers of two. This way, listeners can easily specify multiple **Keywords** to listen for.

Next come some constant declarations for the event identifiers. Using a constant declaration is not required, but it makes it more convenient if both the source and listener need to refer to the identifiers.

Finally, there is the list of events. These are specified with a `void` method that takes any number of arbitrary arguments. These methods are prefaced with an attribute that specifies the ID, the event level, and any keywords you want to apply. You can apply multiple keywords by ORing them together: `Keywords = Keywords.General | Keywords.PrimeOutput`.

There are five event levels:

- **LogAlways**: Always logged, no matter what, regardless of log level specified.
- **Critical**: A very serious error, probably indicating your program cannot safely recover.
- **Error**: A normal error.
- **Warning**: Not quite an error, but someone may want to act on it.
- **Informational**: A purely informational message; does not indicate an error.
- **Verbose**: Should not be logged at all in most situations; useful for debugging specific problems or when running in certain modes.

These levels are cumulative. Specifying a logging level implies that you will receive all events for that level and above. For example, if you specify a log level of `Warning`, you will also get events for `Error`, `Critical`, and `LogAlways`.

The event body is simple. Check to see if events are enabled (this is mostly a performance optimization). If they are, call the `WriteEvent` method (inherited from `EventSource`) with the event ID and your arguments.

Note

Be careful logging `null` values. Earlier versions of `EventSource` system did not know how to interpret them correctly because there is no type information. This is most common with `string` values. The latest version of .NET seems to handle this correctly, silently replacing `null` strings with empty strings. To be extra careful, check for the `null` and supply a reasonable default:

```
[Event(5,
    Level = EventLevel.Informational,
    Keywords = Keywords.General)]
public void Error(string message)
{
    if (IsEnabled())
    {
        this.WriteEvent(5, message ?? string.Empty);
    }
}
```

There are over a dozen overloads of the `WriteEvent` method that take various combinations of parameter types. If none of those overloads satisfy the call, then the default option is `WriteEvent(int eventId, params object[] args)`. This is a method to avoid if you can -it involves allocations and reflection to figure out the correct object types.

To write your events, your code just needs to do something like this:

```
Events.Log.ProcessingStart();  
Events.Log.FoundPrime(7);
```

Consume Custom Events in PerfView

Now that your application is producing ETW events, you can capture these events in any ETW listener utility, such as PerfView, Windows Performance Analyzer, or Windows' built-in utility PerfMon.

To capture custom event sources in PerfView, you need to put the name, preceded by an asterisk (*), in the Additional Providers textbox in the Collect window.

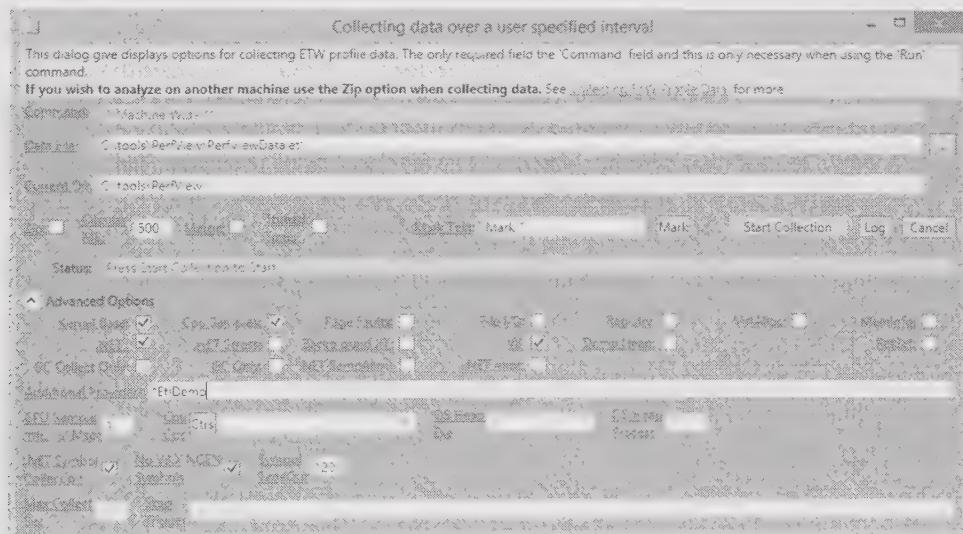


Figure 8.1. PerfView's Collect window, showing where to enter additional ETW providers.

By writing *EtlDemo, you tell PerfView to calculate the GUID automatically, as described earlier in the chapter. You can see more information by clicking on the Additional Providers title link.

Start collecting the samples, run EtlDemo, then press the Stop Collection button. Once the resulting events are recorded, open the raw Events node. You will see a list of all events captured, including these:

- EtlDemo/FoundPrime
- EtlDemo/ManifestData
- EtlDemo/ProcessingStart
- EtlDemo/ProcessingFinish

If you highlight all the events in the list and click the Update button to refresh the view, you will see the list of events you are interested in.

EtlDemo/ProcessingStart	2,701,303	EtlDemo (8296)	ThreadId="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,701,345	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,701,424	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,701,466	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,701,975	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Windows Kernel/PerfInfo/SampleProf	2,702,102	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,702,342	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,702,533	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
EtlDemo/FoundPrime	2,702,564	EtlDemo (8296)	ThreadId="8,556" primeNumber="C
Microsoft-Windows-DotNETRuntime/Method/JittingStarted	2,702,596	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Windows Kernel/PerfInfo/SampleProf	2,703,167	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
Microsoft-Windows-DotNETRuntime/Method/LoadVerbose	2,703,191	EtlDemo (8296)	HasStack="True" ThreadID="8,556"
EtlDemo/FoundPrime	2,704,007	EtlDemo (8296)	ThreadId="8,556" primeNumber="C

Figure 8.2. A sorted list showing Windows, .NET, and application events.

This shows you the custom events in the context of all the other captured events. You can see, for example, the JIT events that precede the FoundPrime events. This hints at the great power you can unleash with some smart ETW analysis. You can do some very detailed performance investigations in the context of your own application's scenarios. You can see a simple example of this later in the chapter.

Create a Custom ETW Event Listener

Most applications will not require you to create your own ETW listener. It is almost always sufficient to define your own events and use an application like PerfView to do the collection and analysis for you. However, you may want to create a listener if you need a custom logger or to perform near real-time event analysis, for example.

In .NET, an event listener is a class that derives from the `EventListener`. To demonstrate multiple ways of handling the event data, I will define a base class for generic handling of listeners.

This class will need to know which events it should listen for and which level and keywords to filter by, so first define a simple structure to encapsulate that information:

```
class SourceConfig
{
    public string Name { get; set; }
    public EventLevel Level { get; set; }
    public EventKeywords Keywords { get; set; }
}
```

Then we can define our listener's constructor as taking a collection of these (one for each event source):

```
abstract class BaseListener : EventListener
{
    List<SourceConfig> configs = new List<SourceConfig>();
    protected BaseListener(
        IEnumerable<SourceConfig> sources)
    {
        this.configs.AddRange(sources);

        foreach (var source in this.configs)
        {
            var eventSource = FindEventSource(source.Name);
            if (eventSource != null)
            {
```

```
        this.EnableEvents(eventSource,
                          source.Level,
                          source.Keywords);
    }
}
}

private static EventSource FindEventSource(string name)
{
    foreach (var eventSource in EventSource.GetSources())
    {
        if (string.Equals(eventSource.Name, name))
        {
            return eventSource;
        }
    }
    return null;
}
```

After saving the sources to its own list, it iterates over them and tries to find an existing `EventSource` that matches the names we want. If it finds one, it subscribes by calling the inherited method `EnableEvents`.

This is not enough, however. It is possible the `EventSource` is created after we set up our listener. For this eventuality, we can override the `OnEventSourceCreated` method and do essentially the same check to see if we are interested in the new `EventSource`.

```
protected override void OnEventSourceCreated(
    EventSource eventSource)
{
    base.OnEventSourceCreated(eventSource);

    foreach (var source in this.configs)
    {
        if (string.Equals(source.Name, eventSource.Name))
        {
            this.EnableEvents(eventSource,
                              source.Level,
                              source.Keywords);
        }
    }
}
```

```
    }  
}
```

The last thing we need to do is handle the `OnEventWritten` event which is called every time a new event is written by the sources for the current listener.

```
protected override void OnEventWritten(  
    EventWrittenEventArgs eventData)  
{  
    this.WriteEvent(eventData);  
}  
  
protected abstract void WriteEvent(  
    EventWrittenEventArgs eventData);
```

In this case, I am just deferring to an abstract method which will do the heavy lifting.

It is common practice to define multiple listener types that expose the event data in different ways. For this sample, I have defined one that writes the messages to the console and another that logs them to a file.

The `ConsoleListener` class looks like this:

```
class ConsoleListener : BaseListener  
{  
    public ConsoleListener(  
        IEnumerable<SourceConfig> sources)  
        :base(sources)  
    {  
    }  
  
    protected override void WriteEvent(  
        EventWrittenEventArgs eventData)  
    {  
        string outputString;  
        switch (eventData.EventId)  
        {  
            case Events.ProcessingStartId:  
                outputString = string.Format("ProcessingStart ({0})",  
                    eventData.EventId);  
        }  
    }  
}
```

```
        break;
    case Events.ProcessingFinishId:
        outputString = string.Format("ProcessingFinish ({0})",
                                      eventData.EventId);
        break;
    case Events.FoundPrimeId:
        outputString = string.Format("FoundPrime ({0}): {1}",
                                      eventData.EventId,
                                      (long)eventData.Payload[0]);
        break;
    default:
        throw new InvalidOperationException("Unknown event");
    }
    Console.WriteLine(outputString);
}
}
```

The `EventId` property is how you determine which event you are looking at. It is not as easy to get the name of the event, unfortunately, but it is possible with some up front work, as you will see later. The `Payload` property provides you an array of the values that were passed into the original event method.

The `FileListener` is only slightly more complicated:

```

switch (eventData.EventId)
{
    case Events.ProcessingStartId:
        output.Append("ProcessingStart");
        break;
    case Events.ProcessingFinishId:
        output.Append("ProcessingFinish");
        break;
    case Events.FoundPrimeId:
        output.AppendFormat("FoundPrime - {0:N0}",
            eventData.Payload[0]);
        break;
    default:
        throw new InvalidOperationException("Unknown event");
}
this.writer.WriteLine(output.ToString());
}

public override void Dispose()
{
    this.writer.Close();

    base.Dispose();
}
}
}

```

This code snippet from EtlDemo demonstrates how to use both listeners and have them listen to different keywords and levels:

```

var consoleListener = new ConsoleListener(
    new SourceConfig []
    {
        new SourceConfig (){
            Name = "EtlDemo",
            Level = EventLevel.Informational ,
            Keywords = Events.Keywords.General}
    });

var fileListener = new FileListener(
    new SourceConfig []
    {
        new SourceConfig (){

```

```
Name = "EtlDemo",
Level = EventLevel.Verbose,
Keywords = Events.Keywords.PrimeOutput}
},
"PrimeOutput.txt");

long start = 1000000;
long end = start + 10000;

Events.Write.ProcessingStart();
for (long i = start; i < end; i++)
{
    if (IsPrime(i))
    {
        Events.Write.FoundPrime(i);
    }
}

Events.Write.ProcessingFinish();
consoleListener.Dispose();
fileListener.Dispose();
```

It first creates the two types of listeners and subscribes them to a different set of events. Then it logs some events and exercises the program.

The console output has just this:

```
ProcessingStart (1)
ProcessingFinish (2)
```

While the output file contains lines like this:

```
2014-03-08-15:21:31.424 - Informational - FoundPrime - 1,000,003
2014-03-08-15:21:31.425 - Informational - FoundPrime - 1,000,033
2014-03-08-15:21:31.425 - Informational - FoundPrime - 1,000,037
```

It is not possible to process events in absolute real-time. For one, the events are received on different threads than those on which they are generated. Secondly, events from multiple event sources can be combined into a single listener, which means there will be queueing at some point.

Get Detailed EventSource Data

You may have noticed something interesting in the previous sections: our own event listener did not know the name of the event it was receiving, but PerfView somehow did. This is possible because every `EventSource` has a manifest associated with it. A manifest is just an XML description of the event source. Thankfully, .NET makes it easy to generate this manifest from an `EventSource` class.

```
string xml =
    EventSource
    .GenerateManifest(typeof(Events), string.Empty);
```

Here is the manifest for our own events defined previously:

```
<instrumentationManifest
  xmlns="http://schemas.microsoft.com/win/2004/08/events">
  <instrumentation
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:win=...>
    <events xmlns=...>
      <provider name="EtlDemo"
        guid="{458d4a63-7cc9-5239-62c4-f8aebbe597ac}"
        resourceFileName=""
        messageFileName=""
        symbol="EtlDemo">
        <tasks>
          <task name="FoundPrime"
            value="65531"/>
          <task name="ProcessingFinish"
            value="65532"/>
          <task name="ProcessingStart"
            value="65533"/>
        </tasks>
        <opcodes>
        </opcodes>
        <keywords>
          <keyword name="General"
            message="$(string.keyword_General)"
```

```
        mask="0x1"/>
    <keyword name="PrimeOutput"
        message="$(string.keyword_PrimeOutput)"
        mask="0x2"/>
</keywords>
<events>
    <event value="1"
        version="0"
        level="win:Informational"
        keywords="General"
        task="ProcessingStart"/>
    <event value="2"
        version="0"
        level="win:Informational"
        keywords="General"
        task="ProcessingFinish"/>
    <event value="3"
        version="0"
        level="win:Informational"
        keywords="PrimeOutput"
        task="FoundPrime"
        template="FoundPrimeArgs"/>
</events>
<templates>
    <template tid="FoundPrimeArgs">
        <data name="primeNumber"
            inType="win:Int64"/>
    </template>
</templates>
</provider>
</events>
</instrumentation>
<localization>
    <resources culture="en-US">
        <stringTable>
            <string id="keyword_General"
                value="General"/>
            <string id="keyword_PrimeOutput"
                value="PrimeOutput"/>
        </stringTable>
    </resources>
</localization>
</instrumentationManifest>
```

.NET is doing some behind-the-scenes magic for you to examine the types you use and generate the resulting manifest. For a more feature-rich logging system, you can parse this XML to get the names of the events and match them to the IDs, as well as the types of all the arguments.

Consuming CLR and System Events

If you want to consume .NET or other Windows system events, there is already a robust library available to use. It is the same event engine that powers PerfView: TraceEvent. This library has been published as a NuGet package named Microsoft.Diagnostics.Tracing.TraceEvent. It provides a simple means to process many types of CLR and OS events in your own code. You could use this to build your own analysis tools or even put it inside your production system itself to allow reaction to GC behavior in near-real-time.

Here is some sample code that listens for all GC Start and Stop events on the system:

```
using Microsoft.Diagnostics.Tracing.Parsers;
using Microsoft.Diagnostics.Tracing.Session;
using System;
using Microsoft.Diagnostics.Tracing.Parsers.Clr;

namespace GCListener
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press Ctrl-C to exit");

            using (var session =
                new TraceEventSession("GCListenSession"))
            {
                Console.CancelKeyPress += (a,b) => session.Stop();
                session.EnableProvider(
                    ClrTraceEventParser.ProviderGuid,
                    Microsoft.Diagnostics.Tracing.TraceEventLevel.Informational,
```

```
(ulong)(ClrTraceEventParser.Keywords.GC));

        session.Source.Clr.GCStart += Clr_GCStart;
        session.Source.Clr.GCStop += Clr_GCStop;

        // This will run until session.Stop is called
        session.Source.Process();
    }
}

private static void Clr_GCStart(
Microsoft.Diagnostics.Tracing.Parsers.Clr.GCStartTraceData
gcStartData)
{
    Console.WriteLine(
$"GCStart: Process: {gcStartData.ProcessID}, " +
$"Gen: {gcStartData.Depth}, Type: {gcStartData.Type}");
}

private static void Clr_GCStop(GCEndTraceData gcEndData)
{
    Console.WriteLine(
$"GCStop: Process: {gcEndData.ProcessID}, " +
$"Gen: {gcEndData.Depth}");
}
```

The output will look something like this:

```
Press Ctrl-C to exit
GCStart: Process: 84592, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 0
GCStart: Process: 84592, Gen: 1, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 1
GCStart: Process: 84592, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 84592, Gen: 0
GCStart: Process: 97844, Gen: 0, Type: NonConcurrentGC
GCStop: Process: 97844, Gen: 0
```

Custom PerfView Analysis Extension

If TraceEvent does not do enough for you, you can actually customize PerfView itself to automate analysis at a higher level, taking advantage of its impressive grouping and folding functionality to generate filtered, relevant stacks for your application.

PerfView ships with its own sample project to get started with this and it is actually built in to PerfView's executable itself. To generate a sample solution, type the following at a command prompt:

```
PerfView.exe CreateExtensionProject MyProjectName
```

This will generate a solution file, project file, and sample source code file, complete with some code samples to get you started. Some examples of what you could do:

- Create a report showing you which assemblies use the most CPU. There is already a generated demo command that does exactly this.
- Automate a CPU analysis to export an XML file showing you the top most expensive stacks in your program, given some kind of criteria.
- Create views with complex folding and grouping schemes that you use frequently.
- Create a view that shows memory allocations for a specific operation in your program, where the operation is defined by your own custom ETW events.

With custom extensions and PerfView's command-line mode (no GUI), you can easily create a scriptable profiling tool that gives you easy-to-analyze reports of the most interesting areas of your application.

Here is a simple example that analyzes the frequency of the FoundPrime events from the EtlDemo sample program. I first captured the events with PerfView in a normal collection, using the *EtlDemo provider.

```
public void AnalyzePrimeFindFrequency(string etlFileName)
{
    using (var etlFile = OpenETLFile(etlFileName))
    {
        var events = GetTraceEventsWithProcessFilter(etlFile);

        const int BucketSize = 10000;
        //Each entry represents BucketSize primes and how
        //long it took to find them
        List<double> primesPerSecond = new List<double>();

        int numFound = 0;
        DateTime startTime = DateTime.MinValue;

        foreach (TraceEvent ev in events)
        {
            if (ev.ProviderName == "EtlDemo")
            {
                if (ev.EventName == "FoundPrime")
                {
                    if (numFound == 0)
                    {
                        startTime = ev.Timestamp;
                    }

                    var primeNumber = (long)ev.PayloadByName("primeNumber");
                    if (++numFound == BucketSize)
                    {
                        var elapsed = ev.Timestamp - startTime;
                        double rate = BucketSize / elapsed.TotalSeconds;
                        primesPerSecond.Add(rate);
                        numFound = 0;
                    }
                }
            }
        }

        var htmlFileName = CreateUniqueCacheFileName(
            "PrimeRateHtmlReport", ".html");
        using (var htmlWriter = File.CreateText(htmlFileName))
    {
```

```
    htmlWriter.WriteLine("<h1>Prime Discovery Rate</h1>");  
    htmlWriter.WriteLine("<p>Buckets: {0}</p>",  
        primesPerSecond.Count);  
    htmlWriter.WriteLine("<p>Bucket Size: {0}</p>", BucketSize);  
    htmlWriter.WriteLine("<p>");  
    htmlWriter.WriteLine("<table border=\"1\">");  
    for (int i = 0; i < primesPerSecond.Count; i++)  
    {  
        htmlWriter.WriteLine(  
            "<tr><td>{0}</td><td>{1:F2}/sec</td></tr>",  
            i,  
            primesPerSecond[i]);  
    }  
    htmlWriter.WriteLine("</table>");  
}  
  
    OpenHtmlReport(htmlFileName, "Prime Discovery Rate");  
}  
}
```

You can run the extension with this command line:

```
PerfView userCommand MyProjectName.AnalyzePrimeFindFrequency  
PerfViewData.etl
```

Everything after the extension name is passed into the method as the arguments.

The output will be a window in PerfView that looks like a web page, with the information you wrote.

Note that the extension capability is not an officially supported API. PerfView's internal API has had breaking changes in the past and likely will so in the future.

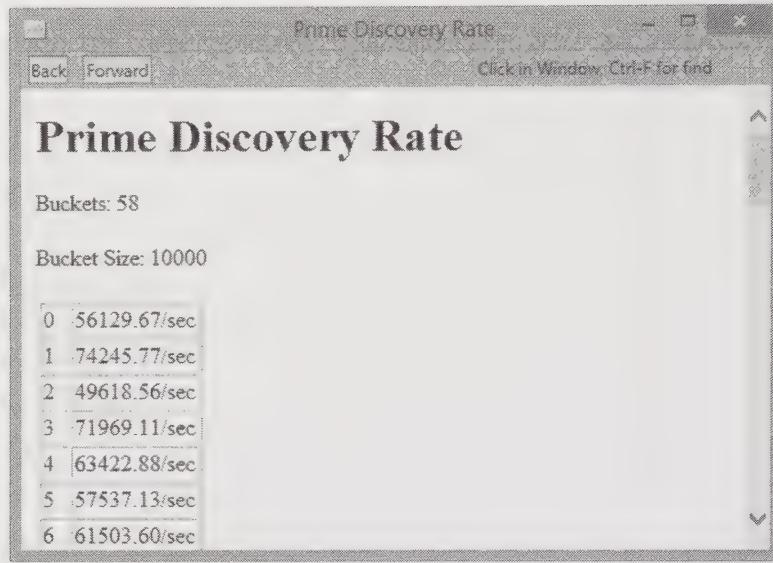


Figure 8.3. The HTML output of our custom ETW analysis.

Summary

ETW events are the preferred method of logging discrete events in your application. They are ideal for both an application log as well as tracking detailed performance information.

In most cases, PerfView or another ETW analysis application will be able to provide all of the investigation tools you need, but if you want custom analysis, use the TraceEvent library or build your own analysis tool with EventListener. To take advantage of advanced grouping and folding capability, build a PerfView extension.

Chapter 9

Code Safety and Analysis

There is a well-worn adage in software engineering that you should first make your code correct, then make it fast. Most of this book has focused strictly on performance, but this chapter is a little bit of an aside into some important topics that, while not strictly related to performance, may help you in your pursuit of high-performance, scalable applications. By undertaking some good practices to ensure the stability and reliability of your code, you free yourself to make more drastic changes for performance's sake. When problems do occur, you will more easily narrow down the location of the issue.

Understanding the OS, APIs, and Hardware

Heavy performance optimization is going to defy any abstractions you want to impose on your software. As mentioned numerous times in this book, you must understand the APIs you call in order to make intelligent decisions about how to use them, or whether to use them at all.

That is not enough, however. Take threading, for example. While various versions of the .NET Framework have added abstractions on top of threads that make asynchronous programming easier, taking advantage of this fully will require you to understand how these features interact with the underlying OS

threads and its scheduling algorithm. The same is true for debugging memory problems. The GC heap is remarkably simple to inspect, but if you have a huge process that loads thousands of types from hundreds of assemblies, you may run into problems outside of the pure managed world, which will require you to understand a process's full memory layout.

Finally, the hardware is just as important. In the chapter on JIT, I mentioned things like locality of reference – putting bits of code and data that are used together physically near each other in memory so that they can be efficiently included in a processor's cache. If you are lucky, your code will target a single hardware platform. If not, then you need to understand how they each execute code differently. You may have different memory limits or different cache sizes, or even more substantial differences such as completely different memory models. Hardware also impacts which kinds of multithreading bugs will be surfaced – some platforms will be more forgiving of synchronization sloppiness, and others will not.

All of this must be balanced with the fact that you cannot optimize everything – you can use profiling to zero-in on the areas of code that need the most attention. You also need to expend effort only in proportion to the true need of performance improvement. On the other hand, the more you understand the building blocks you are working with, the less time and effort you will need to spend in costly performance optimization.

Restrict API Usage in Certain Areas of Your Code

There is no reason why you should allow all components to use the full breadth of every Framework and system API. For example, if you have a strict Task-based processing model, then centralize that functionality and prohibit any other components from accessing anything in the `System.Threading` namespace.

These kinds of rules are particularly important for systems with an extension model. You usually want the platform executing all the hard, dangerous code, while the extensions do simple actions in their respective domains.

One way to prevent dangerous patterns or known-bad performance issues is with static code analysis. There are two primary tools you can freely use to accomplish this in .NET: FxCop and Roslyn Code Analyzers.

FxCop is the older tool and operates on compiled DLLs, meaning it can only understand MSIL. .NET Compiler Code Analyzers (you may have heard the early codename “Roslyn Code Analyzers”), on the other hand, have access to the full syntax tree in the compiler itself, and can run during development. I will show examples of both tools, but I recommend .NET Compiler Code Analyzers for all future development.

Custom FxCop Rules

FxCop is a free static code analysis tool that ships with Visual Studio. It comes with standard rules in categories such as Performance, Globalization, Security, and more, but you can add a library of your own rules. Many of the performance rules we discuss in this book can be represented as FxCop rules, for example:

- Prohibiting use of “dangerous” namespaces
- Banning types or APIs that typically cause LOH allocations
- Banning APIs that have better alternatives such as `TryParse` in lieu of `Parse`
- Finding instances of double-casting
- Finding instances of boxing
- Enforcing specific usage patterns for types (e.g., all `Regex` objects must be `static readonly` and created with the `RegexOptions.Compiled` flag)

Before you start writing rules, keep in mind that FxCop can only analyze IL and metadata. It has no knowledge of C# or any other high-level language. Because of this, you will not be able to enforce static checks that rely on specific

language patterns. Writing your own FxCop rules is easy, but there is little to no official documentation, and you will find yourself relying on analyzing the IL of your programs and making extensive use of IntelliSense to poke through the FxCop API. The more you understand IL, the more complicated rules you can develop.

You will first need to install the FxCop SDK, which is trickier than it should be. If you have Visual Studio Professional or better, then it has been included and rebranded Code Analysis in the IDE, but it is still FxCop underneath. On my machine, the relevant files are located in C:\Program Files (x86)\Microsoft Visual Studio 14.0\Team Tools\Static Analysis Tools\FxCop.

If you cannot get access to the right version of Visual Studio, there are still a few options. The easiest way to download it is from CodePlex at <https://fxcopinstaller.codeplex.com>. If that project has disappeared by the time you read this, then try the Windows 7.1 SDK, which appears to have a broken web installer now, but you can get the ISO image at <https://www.microsoft.com/download/details.aspx?id=8442>. Download it and extract the installer from the archive \Setup\WinSDKNetFxTools\cab1.cab. There is a file inside that archive that begins with the name WinSDK FxCopSetup.exe. Extract that file and rename it to FxCopSetup.exe and you are on your way.

In the source code accompanying this book you will find projects related to FxCop. These are in their own solution file to avoid breaking the build for the rest of the sample projects. FxCopRules contains the rules that will be loaded by the FxCop engine and run against some target assembly. FxCopViolator contains a class with a number of violations that the rules will test against. Follow along with these projects as I explain the various components.

Before you can build the rules, you may need to edit the FxCopRules.csproj file to point to the correct SDK path. The current values should resemble:

```
<PropertyGroup>
  <FxCopSdkDir>C:\...\Microsoft Fxcop 10.0</FxCopSdkDir>
</PropertyGroup>
<ItemGroup>
  <Reference Include="$(FxCopSdkDir)\FxCopSdk.dll" />
  <Reference Include="$(FxCopSdkDir)\Microsoft.CCi.dll" />
</ItemGroup>
```

Update the FxCopSdkDir value to point to the FxCop installation directory (by default in Program Files (x86)), or wherever you have placed the appropriate DLLs.

Next, you will need to create a Rules.xml file that contains the metadata for each rule. Our first rule will look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<Rules FriendlyName="Custom Rules">
  <Rule TypeName="DisallowStaticFieldsRule"
    Category="Custom.Arbitrary"
    CheckId="HP100">
    <Name>Static fields are not allowed</Name>
    <Description>Static fields are not allowed because...
    </Description>
    <Url>http://internaldocumentationsite/FxCop/HP100</Url>
    <Resolution>Make the static field '{0}' either
      readonly or const.
    </Resolution>
    <MessageLevel Certainty="90">Error</MessageLevel>
    <FixCategories>Breaking</FixCategories>
    <Email>feedback@high-perf.net</Email>
    <Owner>Ben Watson</Owner>
  </Rule>
</Rules>
```

Note that the TypeName attribute must match the name of the rule class that we define next. This XML file must be included in the project with the Build Action set to Embedded Resource.

Each rule we define must derive from a class provided by the FxCop SDK and include some common information, such as the location of the XML rules manifest. To make this more convenient, it is a good idea to create a base class for all of your rules that provides this common functionality.

```
using Microsoft.FxCop.Sdk;
using System.Reflection;

namespace FxCopRules
{
  public abstract class BaseCustomRule : BaseIntrospectionRule
```

```
{  
    // The manifest name is the default namespace plus the name  
    // of the XML rules file, without the extension.  
    private const string ManifestName = "FxCopRules.Rules";  
  
    // The assembly where the rule manifest is  
    // embedded (the current assembly in our case).  
    private static readonly Assembly ResourceAssembly =  
        typeof(BaseCustomRule).Assembly;  
  
    protected BaseCustomRule(string ruleName)  
        : base(ruleName, ManifestName, ResourceAssembly)  
    {  
    }  
}  
}
```

Next, define a class that derives from `BaseCustomRule` that will be for a specific violation you want to check. The first example will disallow all `static` fields, but allow `const` and `readonly` fields.

```
public class DisallowStaticFieldsRule : BaseCustomRule  
{  
    public DisallowStaticFieldsRule()  
        : base(typeof(DisallowStaticFieldsRule).Name)  
    {  
    }  
  
    public override ProblemCollection Check(Member member)  
    {  
        var field = member as Field;  
        if (field != null)  
        {  
            // Find all static data that isn't const or readonly  
            if (field.IsStatic && !field.IsInitOnly && !field.IsLiteral)  
            {  
                // field.FullName is an optional argument that will be  
                // used to format the Resolution string's {0} parameter.  
                var resolution = this.GetResolution(field.FullName);  
                var problem = new Problem(resolution,  
                    field.SourceContext);  
                this.Problems.Add(problem);  
            }  
        }  
    }  
}
```

```
        }  
    }  
    return this.Problems;  
}  
}
```

The `BaseCustomRule` class provides a number of virtual `Check` method overrides with various types of arguments which you can override to provide your functionality. By default, these methods do nothing. IntelliSense is your friend while writing FxCop rules, and it reveals the following `Check` methods:

- Check(ModuleNode moduleNode)
 - Check(Parameter parameter)
 - Check(Resource resource)
 - Check(TypeNode typeNode)
 - Check(string namespaceName, TypeNodeCollection types)

You can also examine individual lines of IL code from any method. Here is a rule that prohibits string case conversion.

```
public class DisallowStringCaseConversionRule : BaseCustomRule
{
    public DisallowStringCaseConversionRule()
        : base(typeof(DisallowStringCaseConversionRule).Name)
    { }

    public override ProblemCollection Check(Member member)
    {
        var method = member as Method;
        if (method != null)
        {
            foreach (var instruction in method.Instructions)
            {
                if (instruction.OpCode == OpCode.Call
                    || instruction.OpCode == OpCode.Calli
                    || instruction.OpCode == OpCode.Callvirt)
                {
                    if (instruction.Operand is String)
                    {
                        if (instruction.Operand.ToString().Length > 1)
                        {
                            if (!instruction.Operand.ToString().Contains("System.String"))
                            {
                                yield return new Problem("String conversion rule violation", instruction);
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
        {
            var targetMethod = instruction.Value as Method;
            if (targetMethod != null
                && (targetMethod.FullName == "System.String.ToUpper"
                    || targetMethod.FullName == "System.String.ToLower"))
            {
                var resolution = this.GetResolution(method.FullName);
                var problem = new Problem(resolution,
                                            method.SourceContext);
                this.Problems.Add(problem);
            }
        }
    }

    return this.Problems;
}
```

For a final example, look at a different way to tell FxCop to traverse the code. In addition to the `Check` methods described previously, you can override dozens of `Visit*` methods. These are called in a recursive descent through every node in the program graph, starting at the node you pick. You override just the `Visit` methods you need. Here is an example that uses this to add a rule against instantiating a `Thread` object:

```
public class DisallowThreadCreationRule : BaseCustomRule
{
    public DisallowThreadCreationRule()
        : base(typeof(DisallowThreadCreationRule).Name) { }

    public override ProblemCollection Check(Member member)
    {
        var method = member as Method;
        if (method != null)
        {
            VisitStatements(method.Body.Statements);
        }

        return base.Check(member);
    }
}
```

```
public override void VisitConstruct(Construct construct)
{
    if (construct != null)
    {
        var binding = construct.Constructor as MemberBinding;
        if (binding != null)
        {
            var instanceInitializer =
                binding.BoundMember as InstanceInitializer;
            if (instanceInitializer.DeclaringType.FullName
                == "System.Threading.Thread")
            {
                var problem = new Problem(this.GetResolution(),
                    construct.SourceContext);
                this.Problems.Add(problem);
            }
        }
    }

    base.VisitConstruct(construct);
}
}
```

To use these rules in Visual Studio, build the FxCopRules.dll and copy it to your FxCop installation's Rules folder (mine is C:\Program Files (x86)\Microsoft Visual Studio 14.0\Team Tools\Static Analysis Tools\FxCop\Rules). In Visual Studio, go to another project's properties (you can test this on the FxCopViolator sample project) and view the Code Analysis tab. Under Rule Set, you can select a custom rule set, or create your own that includes whatever rules you want.

Now, when you build a project with the appropriate rules selected, you should see some build messages indicating violations of your custom rules. Just like any standard build or code analysis rule, you can double-click these and jump straight to the source.

The sample project also includes an example of how to run FxCop with custom rules from the command line:

```
>"C:\Program Files (x86)\Microsoft Fxcop 10.0\FxCopCmd.exe"
```

CHAPTER 9. CODE SAFETY AND ANALYSIS

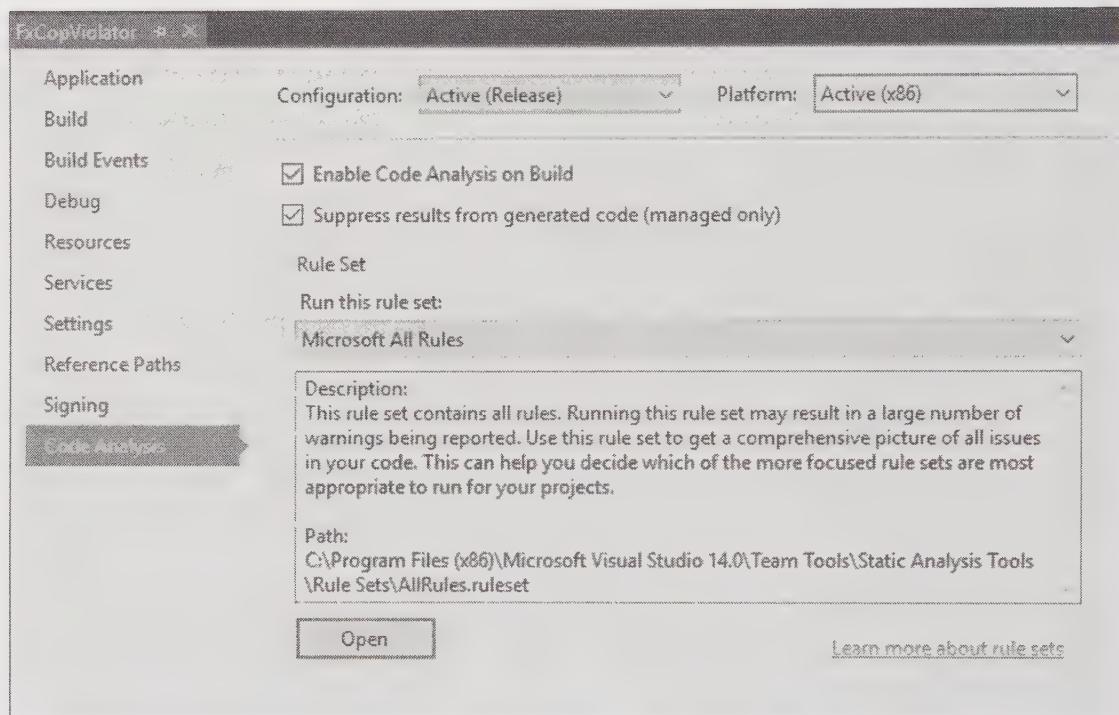


Figure 9.1. You can customize the rules you want to enforce in Visual Studio via the Code Analysis tab in a project's properties.

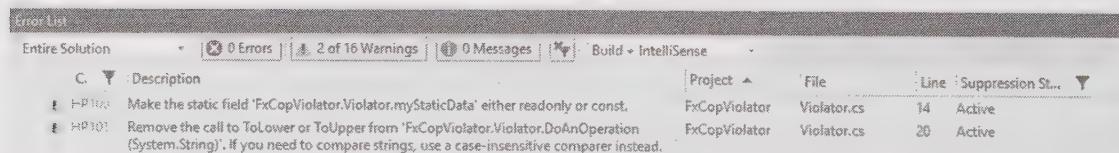


Figure 9.2. Custom FxCop warnings will show up in the Error List pane, just like other build and code analysis issues.

```
/out:.\\FxCopOutput.xml /rule:FxCopRules.dll  
/file:FxCopViolator.dll
```

```
Microsoft (R) FxCop Command-Line Tool, Version 14.0 (14.0.25420.1)  
Copyright (C) Microsoft Corporation, All Rights Reserved.
```

```
Loaded fxcoprules.dll...  
Loaded FxCopViolator.dll...  
Initializing Introspection engine...  
Analyzing...  
Analysis Complete.  
Writing 3 messages...  
Writing report to .\\FxCopOutput.xml...  
Done:00:00:00.8200342
```

It is pretty straightforward once you learn how it works. The biggest obstacle to creating your own rules is really the paucity of official documentation. To learn more about custom FxCop rules, read an excellent walkthrough by Jason Kresowaty at <http://www.binarycoder.net/fxcop/>.

.NET Compiler Code Analyzers

In contrast to FxCop's limited understanding of code, .NET Compiler Code Analyzers can not only analyze high-level language code instead of IL, they can do so from within the Visual Studio IDE and they can even suggest and perform code edits. These types of analyzers replace, and are far more powerful than, FxCop. This section will walk through creating a couple rules, one that requires `static` fields to also be marked `readonly`, and another one to warn against calling `String.ToLower` and `String.ToUpper`.

In Visual Studio 2015, you will need to install the “Visual Studio Extensibility Tools” component from the installer. In Visual Studio 2017, it has a slightly different name: “Visual Studio extension development.”

In both versions, you will need to install “.NET Compiler Platform SDK,” which you can do from Visual Studio directly by going to File | New Project | Visual C# | Extensibility and selecting the “Download the .NET Compiler Platform

SDK” in the list of project types. Once the installation is completed, restart Visual Studio. Then, to create your own analyzer, you can choose “Analyzer with Code Fix (NuGet + VSIX)” in the New Project selection window.

The example produced here is available in the CodeAnalyzers sample solution in the accompanying source code. There are three projects:

- **SampleCodeAnalyzer.csproj**: Contains the actual analysis and fixer code.
- **SampleCodeAnalyzer.Test.csproj**: Unit tests to exercise your code without needing to debug in Visual Studio.
- **SampleCodeAnalyzer.Vsix**: The Visual Studio add-on project that enables your analyzer to be hosted in Visual Studio.

To test out the analyzer, make sure the Vsix project is selected as default, and hit F5. This will start another instance of Visual Studio with the analyzer loaded. You can create a new project in here to test out the code analysis.

Our first code analyzer will detect any `static` field and recommend it be marked `readonly`. In addition, it will contain a fix provider to actually do this for you.

The contents of `StaticFieldAnalyzer.cs` show how this is done:

```
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;

namespace SampleCodeAnalyzer
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class StaticFieldAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId = "StaticFieldAnalyzer";

        private static readonly LocalizableString Title =
            new LocalizableResourceString(
                nameof(Resources.AnalyzerTitle),
                Resources.ResourceManager,
```

```
        typeof(Resources));  
  
private static readonly LocalizableString MessageFormat =  
    new LocalizableResourceString(  
        nameof(Resources.AnalyzerMessageFormat),  
        Resources.ResourceManager,  
        typeof(Resources));  
  
private static readonly LocalizableString Description =  
    new LocalizableResourceString(  
        nameof(Resources.AnalyzerDescription),  
        Resources.ResourceManager,  
        typeof(Resources));  
  
private const string Category = "Thread Safety";  
  
private static DiagnosticDescriptor Rule =  
    new DiagnosticDescriptor(DiagnosticId,  
                            Title,  
                            MessageFormat,  
                            Category,  
                            DiagnosticSeverity.Info,  
                            isEnabledByDefault: true,  
                            description: Description);  
  
public override ImmutableArray<DiagnosticDescriptor>  
    SupportedDiagnostics  
{  
    get  
    {  
        return ImmutableArray.Create(Rule);  
    }  
}  
  
public override void Initialize(AnalysisContext context)  
{  
    context.RegisterSymbolAction>AnalyzeFieldSymbol,  
        SymbolKind.Field);  
}  
  
private void AnalyzeFieldSymbol(  
    SymbolAnalysisContext context)  
{
```

```
    IFieldSymbol field = (IFieldSymbol)context.Symbol;
    if (field.IsStatic && !field.IsReadOnly)
    {
        var diagnostic = Diagnostic.Create(
            Rule,
            field.Locations[0],
            field.Name);

        context.ReportDiagnostic(diagnostic);
    }
}
```

The fields at the top of the class are standard boilerplate metadata you should customize for each rule. The project template puts these strings into Resources.resx by default to make them easier to localize, but there is no requirement that you do so.

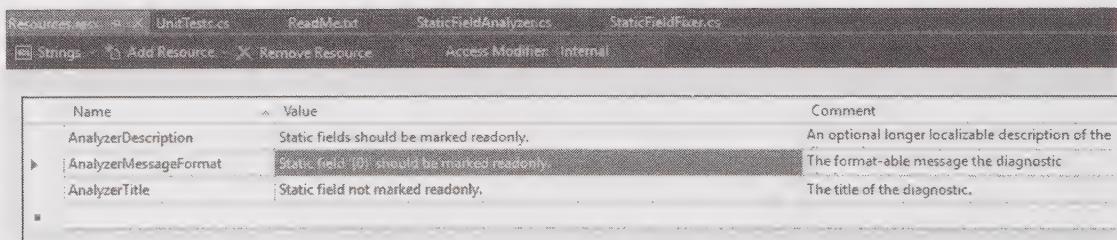


Figure 9.3. The Resources.resx file contains the localizable strings for your code analyzer.

The `Initialize` method tells Visual Studio what kinds of things you want to analyze. In this case, we are analyzing just fields, but we will see another option later. The `AnalyzeFieldSymbol` method is where the action is. This method is called for every symbol of the type we asked for. The code checks to see if the field is `static`, but not `readonly`, and if so it reports a new diagnostic, which will be surfaced in the user interface via a green squiggly line underneath the problematic symbol. The squiggly line is green because we marked the rule as `DiagnosticSeverity.Info`.

```
class Program
{
    private static string staticField;
    ^ (field) string Program.staticField

    Static field 'staticField' should be marked readonly.

    The field 'Program.staticField' is never used

    Show potential fixes (Ctrl+.)
```

Figure 9.4. The code analysis rule causes a “squiggly” line to appear underneath the syntax in question.

In some cases, you may be able to automatically fix the code according to your recommendation. Doing so is not required, but it is nice if you can do it. The `StaticFieldFixer.cs` file contains this code to implement our `readonly` fix automatically.

```
using System.Collections.Immutable;
using System.Composition;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SampleCodeAnalyzer
{
    [ExportCodeFixProvider(LanguageNames.CSharp,
        Name = nameof(StaticFieldFixer)),
     Shared]
    public class StaticFieldFixer : CodeFixProvider
    {
        private const string title = "Make readonly";

        public sealed override ImmutableArray<string>
            FixableDiagnosticIds
```

```
{  
    get  
    {  
        return ImmutableArray.Create(  
            StaticFieldAnalyzer.DiagnosticId);  
    }  
}  
  
public sealed override FixAllProvider GetFixAllProvider()  
{  
    return WellKnownFixAllProviders.BatchFixer;  
}  
  
public sealed override async Task RegisterCodeFixesAsync(  
    CodeFixContext context)  
{  
    var root = await context.Document.GetSyntaxRootAsync(  
        context.CancellationToken).ConfigureAwait(false);  
  
    var diagnostic = context.Diagnostics.First();  
    var diagnosticSpan = diagnostic.Location.SourceSpan;  
  
    // Find the type declaration identified  
    // by the diagnostic.  
    var declaration = root.FindToken(diagnosticSpan.Start)  
        .Parent.AncestorsAndSelf()  
        .OfType<FieldDeclarationSyntax>()  
        .First();  
  
    // Register a code action that will invoke the fix.  
    context.RegisterCodeFix(  
        CodeAction.Create(  
            title: title,  
            createChangedDocument:  
                c => MakeReadOnlyAsync(context.Document,  
                    declaration,  
                    c),  
            equivalenceKey: title),  
        diagnostic);  
}  
  
private async Task<Document> MakeReadOnlyAsync(  
    Document document,
```

```
        FieldDeclarationSyntax fieldDecl,
        CancellationToken cancellationToken)
    {
        // Find the field and update its modifiers
        var newFieldDecl = fieldDecl.AddModifiers(
            SyntaxFactory.Token(
                SyntaxKind.ReadOnlyKeyword));

        var root = await document.GetSyntaxRootAsync();

        // Replace the node with a new one
        var newRoot = root.ReplaceNode(fieldDecl,
                                       newFieldDecl);

        var newDocument = document.WithSyntaxRoot(newRoot);

        // Return the new solution with the
        // now-uppercase type name.
        return newDocument;
    }
}
```

The `FixableDiagnosticIds` property associates the analyzer with this fixer so Visual Studio knows what actions it can take. The `RegisterCodeFixesAsync` method finds the diagnostic and registers a delegate to be called to fix the code. The `MakeReadOnlyAsync` method is what does the real work. It returns a `Document` object that represents the new code document, after the fixes it generates. In this case, it takes the field declaration and adds `readonly` to the list of modifiers. The `SyntaxFactory` class contains a wealth of options to create new pieces of code.

This code works by modifying the individual nodes of the document's syntax tree. The syntax tree is immutable, so doing any modifications causes a new version of the object to be created and returned to you. The `MakeReadOnlyAsync` method successively retrieves new versions of the field, the syntax node, and the document.

Look at another trivial example of an analyzer that recommends that you do not call the methods `String.ToLower` and `String.ToUpper`.

CHAPTER 9. CODE SAFETY AND ANALYSIS

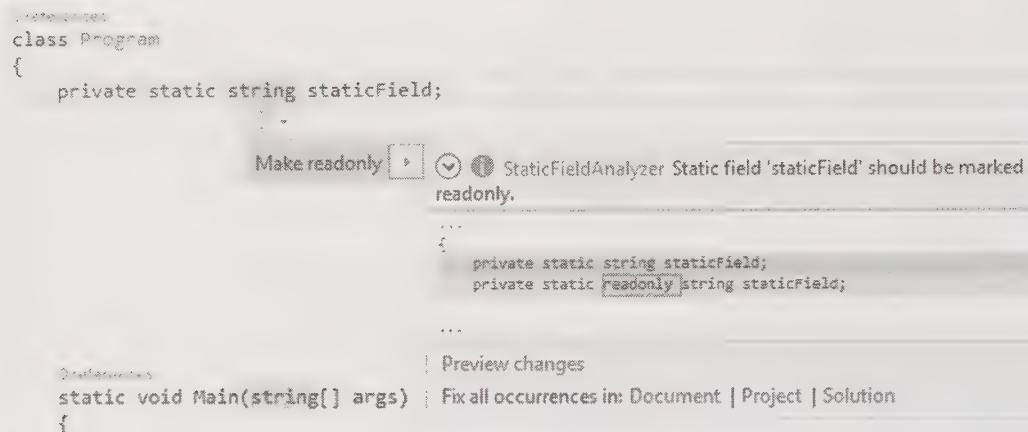


Figure 9.5. Clicking on the *Code Tips* icon brings up an option to automatically make the fix, with a preview of what will be changed.

```
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace SampleCodeAnalyzer
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class StringToUpperToLowerAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId =
            "StringToUpperToLowerAnalyzer";

        private static readonly LocalizableString Title =
            new LocalizableResourceString(
                nameof(Resources.ToUpperToLowerAnalyzerTitle),
                Resources.ResourceManager,
                typeof(Resources));

        private static readonly LocalizableString MessageFormat =
            new LocalizableResourceString(
                nameof(Resources.ToUpperToLowerAnalyzerMessageFormat),
                Resources.ResourceManager,
```

```
    typeof(Resources));  
  
private static readonly LocalizableString Description =  
    new LocalizableResourceString(  
        nameof(Resources.ToUpperToLowerAnalyzerDescription),  
        Resources.ResourceManager,  
        typeof(Resources));  
  
private const string Category = "Performance";  
  
private static DiagnosticDescriptor Rule =  
    new DiagnosticDescriptor(DiagnosticId,  
                            Title,  
                            MessageFormat,  
                            Category,  
                            DiagnosticSeverity.Warning,  
                            isEnabledByDefault: true,  
                            description: Description);  
  
public override ImmutableArray<DiagnosticDescriptor>  
    SupportedDiagnostics  
{  
    get  
    {  
        return ImmutableArray.Create(Rule);  
    }  
}  
  
public override void Initialize(AnalysisContext context)  
{  
    context.RegisterSyntaxNodeAction(  
        AnalyzeNode,  
        SyntaxKind.InvocationExpression);  
}  
  
private void AnalyzeNode(  
    SyntaxNodeAnalysisContext context)  
{  
    var invocationExpression =  
        (InvocationExpressionSyntax)context.Node;  
    var memberAccessExpression =  
        invocationExpression.Expression  
            as MemberAccessExpressionSyntax;
```

```

        var memberName =
            memberAccessExpression?.Name.ToString();
        if (memberName == "ToUpper"
            || memberName == "ToLower")
        {
            var diagnostic = Diagnostic.Create(
                Rule,
                memberAccessExpression.GetLocation());

            context.ReportDiagnostic(diagnostic);
        }
    }
}

```

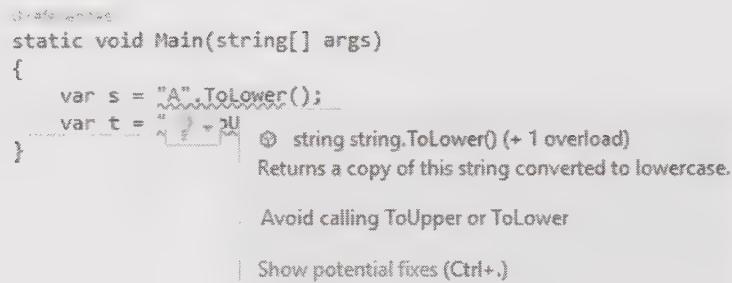


Figure 9.6. With code analyzers, you can encourage good programming practices with direct feedback in Visual Studio.

One additional tool that can help as you develop analyzers is the Syntax Visualizer. You can install this in Visual Studio, via Tools | Extensions and Updates. Search for .NET Compiler Platform SDK, which includes the Syntax Visualizer. Once installed, open it from View | Other Windows | Syntax Visualizer. When it is open, you can click anywhere in a code file and see the syntax tree updated.

You can do almost anything in code analyzers. They are very flexible and allow you almost unlimited freedom to analyze your own code. Some examples:

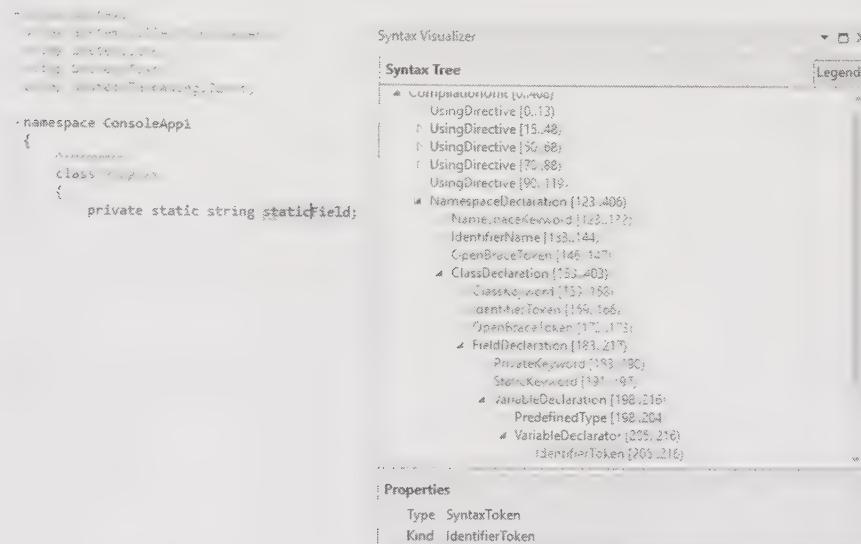


Figure 9.7. The Syntax Visualizer tool can help you understand code's structure as you develop analysis rules.

- Parse, compile, and pre-execute the code, as you type it, analyzing the results according to whatever rules you want.
- Analyze string literals for semantic correctness (such as the illegal presence of credentials, or regular expression validity).
- Enforce coding guidelines particular to your team or company.
- Enforce performance standards specific to your product.

As you gain experience and deeper understanding of both your own code and the way its syntax is represented by the compiler, you can iterate your analyzers to provide more complete and robust analysis.

Thankfully, you do not have to write everything yourself. There are a number of existing code analyzers out there for you to reuse, including:

- Roslyn Analyzers: These ship with the compiler. Browsing the source code will teach you a lot about writing analyzers.

- Clr Heap Allocation Analyzer: Highlights every source of heap allocation, including implicit allocations.
- StyleCopAnalyzers: Coding style recommendations.
- and many more...

These should all be easily searchable with your favorite search engine.

Centralize and Abstract Performance-Sensitive and Difficult Code

You should keep as much performance-sensitive code as possible in one place for easy maintenance, preferably behind APIs that the rest of your application uses. For example, if your application downloads files via HTTP, you could wrap this in an API that exposes only the parts of downloading that the rest of your program needs to know (e.g., the URL you are requesting and the downloaded content). The API manages the complexity of the HTTP call and your entire application goes through that API every time it needs to make an HTTP call. If you discover a performance problem with downloading, or need to enforce a download queue, or any other change, it is trivial to do behind the API. Remember that those APIs need to maintain the asynchronous nature of the operation.

Isolate Unmanaged and Unsafe Code

For many reasons, you should move away from unmanaged code if at all possible. As discussed in the introduction, the benefits of unmanaged code are often exaggerated, but the danger of memory corruption is all too real.

That said, if you have to keep any unmanaged code around (say, to talk to a legacy system, and it is too expensive to move the entire interface to the managed world), then isolate it well. There are many ways to do the isolation, but you absolutely want to avoid having random bits of your system call into unmanaged code all over the place. This is a recipe for chaos.

Ideally, split the unmanaged code into its own process to provide strict OS-level isolation. If that is not possible and you need the unmanaged code to be loaded into the same process, try to keep it in as few DLLs as possible and have all calls to it go through a centralized API that can enforce standard safeguards.

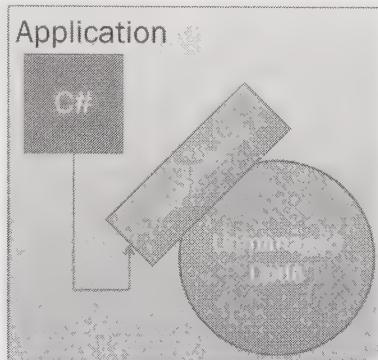


Figure 9.8. Isolate unsafe code into known regions in your application. Subject them to extra scrutiny.

Unmanaged code introduces significant risk into your process. Any bugs that corrupt memory on the unmanaged side of the process can corrupt anything in the process, including memory on the managed side. You lose the safety guarantees that the CLR provides.

Treat managed code that is marked unsafe exactly like unmanaged code and isolate it to as small a scope as you can. You will also need to enable unsafe code in the project settings.

Prefer Code Clarity to Performance Until Proven Otherwise

Code readability and maintenance is more important than performance until proven otherwise. If you find you do need to make deep changes for performance reasons, do it in a way that is as transparent to the code above it as possible.

Once you do make the code worse to read in favor of performance, make sure you document in the code why you are doing it so that someone does not come by after you and “clean up” your elegant optimization by making it simpler.

Summary

To ensure your code is safe, you must understand the implementation details at all levels. Isolate your riskiest code, especially native or unsafe code, to specific modules to limit exposure. Ban problematic APIs and coding patterns and enforce reasonable code standards to encourage safe practices. Enforce these practices with code analyzers or other static analysis build tools. Do not sacrifice code clarity or maintainability for performance unless it is particularly justified.

Chapter 10

Building a Performance-Minded Team

Most interesting software products are not built by a single person. Chances are, you are part of a team trying to build some useful and well-performing software. If you are the performance guru on your team, or even if you are not, then there are things you can do to encourage others to be like-minded as well.

Much of the advice in this chapter presupposes an organization that understands software engineering as a true engineering effort. Unfortunately, many people find themselves in less-than-ideal circumstances. If this is you, do not despair. Perhaps some of the advice in this chapter can help you improve the level of engineering appreciation and competence in your company.

Understand the Areas of Critical Performance

You cannot optimize everything, almost by definition. This goes back to the first principles we discussed in Chapter 1 about measurement and finding the areas of critical performance. As a team, you will need to build consensus on which areas are critical and which can be left alone.

As engineers, we all should have pride in our work and make it the best it can be, and in no area of code should we completely slack off. However, the realities of business dictate limited time and human resources with which to accomplish the necessary work. Given these realities, you should take time to understand where the critical areas of the system are -remember **Measure, Measure, Measure!** and make sure those receive a larger portion of careful attention to detail.

Performance is not the only metric by which you need to judge code. Maintainability, scalability, security, configurability, and other important factors must also guide your decision-making. However, of the items in that list, I suspect that performance measurement and tuning will take more of your time on a continual basis.

Effective Testing

This is not a testing book, but it should go without saying that having effective tests at all levels will greatly increase your confidence in making significant changes to code. If you have unit tests with high code coverage, drastically changing a central algorithm or data structure to be much more efficient should not fill you with dread.

More to the point of this book, if performance is critical to you, then you should be tracking it with the tools and techniques mentioned in this book. Just like you have functional tests, you can have performance tests. They may be as simple as tracking how many operations a component can do in a second, to as complicated as the performance across thousands of metrics between your pre-release server farm and the production server farm.

Performance test failures should be treated as seriously as functional test failures and should be ship-blockers. You will likely find that building reliable, repeatable performance tests is far more difficult than functional tests. So much of performance is intertwined with the state of machines, other software running on the machine, history of the running process, and infinitely more variables. There are two basic approaches to handling this noise that will creep in:

1. **Remove noise:** Have clean machines, restart them before testing, control all the processes running, control for hardware differences, and more. This approach is required if you are comparing performance between two machines. It is also important to calibrate this by running a baseline test on the control and treatment machines to ensure that when the hardware and OS configuration are exactly equal, running the same test on both machines yields similar numbers. It is quite common for machines that spec out the same to have differences creep in that affect test numbers in ways that will throw you off.
2. **Run tests at a large scale:** If it is not practical to eliminate all the noise, then ignore it and run your tests on a scale that is sufficient to reduce the significant sources of noise. This can be quite expensive, especially for larger infrastructures. You may need dozens, hundreds, even thousands of machines to get a truly statistically significant result. If you cannot scale out the hardware, you can scale out in time and rerun tests for hundreds of iterations, but this does not account for as many variables.
3. **Test performance in production:** In many ways, this is the preferred option. There are many types of performance problems that will never manifest themselves in non-production environments, or at lesser scales. Monitoring production performance has to be done anyway, so why not expand that infrastructure to include A/B testing, request sampling, continuous profiling, rollout monitoring, and other tools to let you see how performance is changing over time? Testing of any kind in production requires a confident team and organization, operational experience and skills, and a lot more infrastructure around safety, rollbacks, alerting, and more.

Either way, you will need to engage in A/B testing; that is, comparing the performance of one build against another in as an ideal scenario as you can manage.

Performance Infrastructure and Automation

You will probably need to build some custom infrastructure, tools, and automation support to gather performance data for you, but all the tools to read the metrics are described in this book. Thankfully, nearly all useful performance tools are scriptable in some way.

There are many ways to track performance and you will have to decide the best way for your product. Some ideas include:

- **PerfMon:** If all of your data is represented by performance counters and runs on a single machine, this may be sufficient.
- **Performance counter aggregation:** If you run on multiple machines, you probably need to aggregate the counter information into a centralized database. This has the advantage of storing performance data for historical analysis.
- **Benchmarks:** Your application processes a standard set of data and the resulting performance metrics are compared with historical results. Benchmarks are useful, but you must be careful with their historical validity as scenarios change. You should update your benchmarks when production data changes so that comparisons remain valid.
- **Automated profiling:** Perform random profiling of CPU and memory allocation on either test or real data.
- **Alerts that fire on performance data:** For example, send off an automated alert to a support team if a server CPU gets too high for too long, or the number of tasks queued is increasing.
- **Automated analysis of ETW events:** This can get you some very nitty-gritty detail that performance counters will miss.

The things you do now to build a performance infrastructure will pay huge dividends in the future as performance maintenance becomes mostly automated. Building this infrastructure is usually far more important than fixing any arbitrary actual performance problem because a good infrastructure will be able to

find and surface performance problems much earlier than any manually driven process. A good infrastructure will prevent you from being surprised by bad performance at an inconvenient time. It will also serve as a great regression testing service to ensure that you always maintain an acceptable level of performance as other development occurs.

The most important part of your infrastructure is how much human involvement is necessary. If you are like most software engineers, there is far more work than time to do it in. Relying on manual performance analysis means it will often not get done. Thus, automation is the key to an effective performance strategy. An investment up front will save countless hours day after day. A good time-saving device can be as simple as a script that runs the tools and generates the report for you on demand, but it will likely need to scale with the size of your application. A large server application that runs in a data center will need different kinds of performance analysis than a desktop application, and a more robust performance infrastructure.

Think of what the ideal infrastructure is for your setting and start building it. Treat it as a first-class project in every way, with milestones, adequate resourcing, and design and code reviews. Iterate on it in a way that makes the infrastructure usable in some way very early on and gradually add automation features over time.

Selling these ideas to management can be an uphill battle in some cases. In that case, consider the following tips:

- **Return on Investment:** Management understand money. Expenditure of effort (money) now means less effort (money) later.
- **Total Cost of Ownership:** Again, it comes down to money and time. If investments can reduce expenditures in other areas, then it can become worth it.
- **Stay High-Level:** Talk in language that management understands. If they care about technical details, then discuss them as appropriate, but otherwise stick to the issues they are concerned with.

- **Accept Politics:** Doing the right thing is not always inevitable. Often, factors outside of resourcing or technical factors can influence decisions. Be aware of the politics and try to account for them. Compromise may be necessary.
- **Multi-party Buyoff:** The more people, from more diverse backgrounds or positions, that are behind your proposals, the more likely it will be taken seriously.
- **Evidence:** If you can point to concrete incidents, data, case studies, or previous disasters, as evidence for your designs, then do so.

Believe Only Numbers

In many teams where performance is an afterthought, performance improvements are often pursued only when problems occur that are serious enough to affect the end-user. This means there is an ad-hoc process that boils down to:

User: Your application is too slow!

Dev: Why?

User: I don't know! Just fix it!

Dev: <Does something to make it faster, maybe?>

You do not ever want to have this conversation. Always have the numbers available measuring whatever it is you are being judged by. Have data to back up literally everything you do. People have infinitely more credibility when backed up by numbers and charts. Of course, you will want to make sure those numbers are correct before you publicly rely on them!

Another aspect of numbers is ensuring that your team has official, realistic, and tangible goals. In the example above, the only “metric” was “faster.” This is an unofficial, fuzzy, and mostly worthless goal to have. Make sure you have real, official performance goals and get your leadership chain to sign off on them. Have deliverables for specific metrics. Let it be known that you will not accept unofficial pressure for better performance after the fact.

For more information about setting good performance goals, see Chapter 1.

Effective Code Reviews

No developer is perfect and having multiple sets of eyes can dramatically improve the quality of anyone's code. All code should probably go through some code review process, whether it is via a system of reviewing diffs via email, or a formal sit-down with the whole team.

Recognize that not all code is equally critical to your business. While it might be tempting to say that all code should adhere to the highest standards, that might be a high bar to reach at first. You may want to consider a special category of code reviews for software that has an especially high business impact, where a mistake in functionality or performance can cost real money (or someone's job!). For example, you could require two developers to sign off before code submission, requiring one to be a senior developer or a subject matter expert. For large, complicated code reviews, put everyone into a room with their own laptops and someone projecting and have at it. The exact process depends on your organization, resources, and culture, but develop a process and go with it, modifying as necessary.

It may be helpful to have code reviews that focus on particular aspects of the code, such as functional correctness, security, or performance. You may ask specific people to comment only on their areas of expertise.

Effective code reviewing does not equate to nitpicking. Stylistic differences should often be ignored. Sometimes even larger issues should be glossed over if it does not truly matter and there are more important things to focus on. Just because code is different than how you would write it, does not mean it is necessarily worse. Nothing is more frustrating than going into a code review expecting to dissect some tricky multi-threaded code and instead spending all the time arguing about correct comment syntax or other trivialities. Do not tolerate such wastes of time. Set the expectations for how code reviews should run and enforce it. If there are legitimate standards violations, do not ignore them, but focus on the important things first.

On the other hand, do not accept lame excuses like, “Well, I know that line is inefficient, but does that really matter in the grand scheme of things?” The proper response to this is, “Are you asking how bad you’re allowed to make your code?” You do need to balance overlooking minor issues with the need to create a culture of performance so that next time, the developer does the right thing automatically. There also needs to be evidence of poor performance — either previous experience in similar situations or actual benchmarks. Save the criticism for obvious problems.

Finally, do not buy into the notion of total code “ownership.” Everyone should feel ownership for the entire product. There are no independent, competing kingdoms, and no one should be over-protective of “their” code, regardless of original authorship. Having owners for the purposes of gatekeeping and code reviews is great, but everyone should feel empowered to make improvements to any area of code. Check your ego at the door.

Education

A performance mindset requires training. This can be informal, from a team guru or books such as this one, or formal, with paid classes from a well-known lecturer in the area.

Keep in mind that even those who already know .NET programming will need to change their programming habits once they start acquiring a serious performance mentality.

Likewise, people who are well-versed in C or C++ will need to understand that the rules for achieving good performance are often completely different or backwards from what they thought in the unmanaged code world.

Change can be hard and most people resist it, so it is best to be sensitive when trying to enforce new practices. It is also always important to build leadership support for what you are trying to accomplish.

If you want to kick-start some performance discussions with your peers, here are some ideas:

- Host brown-bag lunch meetings to share what you are learning.
- Start an internal or public blog to share your knowledge or discuss performance issues you have discovered in the products.
- Pick a team member to be your regular performance related reviewer.
- Demonstrate benefits of improving performance with simple benchmarks or proof-of-concept programs.
- Designate someone as a performance specialist who will stay on top of the performance, do code reviews, educate others about good practices, and stay up-to-date on industry changes and the state of the art. If you are reading this, you have already volunteered for this.
- Bring up areas of potential improvement. Tip: It is best to start with your own code first!
- Get your organization to buy copies of this book for everyone. (Shameless plug!)

Summary

Start small when creating a performance mindset in your team. Begin with your own code and take time to understand which areas actually matter for performance. Build an attitude that performance regressions are just as serious as functional failures. Automate as much as possible to reduce the burden on the team. Judge performance metrics on hard numbers, not gut feeling or subjective perception.

Build an effective code review culture that encourages a good coding style, a focus on things that really matter, and collective code ownership.

Recognize that change is hard and you need to be sensitive. Even those familiar with .NET will likely need to change their ways. C++ and Java veterans will not necessarily be great .NET programmers right away.

Find ways to kick-start regular performance conversations and find or create experts to disseminate this information.

Appendix A

Kick-Start Your Application’s Performance

This book goes through hundreds of details that may be a problem in your application, but if you are just getting started, here is a general outline of how you can proceed and analyze your own program’s performance.

Define Metrics

- Define the metrics you are interested in.
- Decide what kind of statistics you need: average, min, max, percentiles, or more complex.
- What are the resource constraints you are operating under? Possible values include, but are not limited to: CPU, memory usage, allocation rate, network I/O, disk usage, disk write rate, to name a few.
- What are the goals for each metric or resource?

Analyze CPU Usage

- Use PerfView or Visual Studio Standalone Profiler to get a CPU profile of your application doing work.
- Analyze the stacks for functions that stand out.
- Is data processing taking a long time?
 - Can you change your data structure to be in a format that requires less processing? For example, instead of parsing XML, use a simple binary serialization format.
 - Are there alternate APIs?
 - Can you parallelize the work with `Task delegates` or `Parallel.For`?

Analyze Memory Usage

- Consider the right type of GC:
 - Server: Your program is the only significant application on the machine and needs the lowest possible latency for GCs.
 - Workstation: You have a UI or share the machine with other important process.
- Profile memory with PerfView:
 - Check results for top allocators – are they expected and acceptable?
 - Pay close attention to the Large Object allocations.
- If gen 2 GCs happen too often:
 - Are there a lot of LOH allocations? Remove or pool these objects.

- Is object promotion high? Reduce object lifetime so that lower generation GCs can collect them. Allocate objects only when needed and **null** them out when no longer needed.
- If objects are living too long, pool them.
- If gen 2 GCs take too long:
 - Consider using GC notifications to get a signal when GC is about to start. Use this opportunity to stop processing.
 - Reduce frequency of full GCs with lower object promotion and fewer LOH allocations.
- If you have high number of gen 0/1 GCs:
 - Look at highest area of allocations in the profile. Find ways to reduce the need for memory allocations.
 - Minimize object lifetime.
- If gen 0/1 GCs have a high pause time:
 - Reduce allocations overall.
 - Minimize object lifetime.
 - Are objects being pinned? Remove these if possible, or reduce the scope of the pinning.
 - Reduce object complexity by removing references between objects.
- If the LOH is growing large:
 - Check for fragmentation with WinDbg or CLR Profiler.
 - Compact the large object heap periodically.
 - Check for object pools with unbounded growth.

Analyze JIT

- If your startup time is long:
 - Is it really because of JIT? Loading application-specific data is a more common cause of long startup times. Make sure it really is JIT.
 - Use PerfView to analyze which methods take a long time to JIT.
 - Use Profile Optimization to speed up JIT on application load.
 - Consider using NGEN.
 - Consider a custom warmup solution that exercises your code.
- Are there methods showing up in the profile that you would expect to be inlined?
 - Look at methods for inlining blockers such as loops, exception handling, recursion, and more.

Analyze Asynchronous Performance

- Use PerfView to determine if there are a high number of contentions.
 - Remove contentions by restructuring the code to need fewer locks.
 - Use `Interlocked` methods or hybrid locks where necessary.
- Capture Thread Time events with PerfView to see where time is being spent. Analyze these areas of the code to ensure that threads are not blocking on I/O.
 - You may have to significantly change your program to be more asynchronous at every level to avoid waiting on `Task` objects or I/O.
 - Ensure you are using asynchronous stream APIs.

- Does your program take a while before it starts using the thread pool efficiently? This can manifest itself as initial slowness that goes away within a few minutes.
 - Make sure the minimum thread pool size is adequate for your workload.

Appendix B

Higher-Level Performance

This book is primarily concerned with helping you understand the basics of performance from a nuts-and-bolts perspective. It is critical to understand the cost of your building blocks before putting together larger applications. Everything covered so far in this book applies to most .NET application types, including those discussed in this appendix.

This appendix will take a step higher and give you a few brief tips for popular types of applications. I will not cover these topics in nearly as much detail as in the rest of this book, so think of this as a general overview to inspire further research. For the most part, these tips are not related to .NET, per se, but are architecture-, domain-, or library-specific.

ASP.NET

- Disable unneeded HTTP modules.
- Remove unused View Engines.
- Do not compile as debug in production (Check for presence of `<compilation debug="true"/>`).
- Reduce roundtrips between browser and server.

- Ensure page buffering is enabled (it is enabled by default).
- Understand and use caches aggressively:
 - `OutputCache`: Caches page output.
 - `Cache`: Caches arbitrary objects however you desire.
- Understand how large your pages are (from a client perspective).
- Remove unnecessary characters and whitespace from pages.
- Use HTTP compression.
- Use client-side validation to save on round-trips, and server-side validation to verify.
- Disable or limit `ViewState` to small objects. If you must use it, compress it.
- Turn off session state if it is not needed.
- Do not use the `Page.DataBind` method.
- Pool connections to backend servers, such as databases.
- Precompile the web-site.
- Check `Page.IsPostBack` property to run code that only needs to happen once per page, such as for initialization.
- Prefer `Server.Transfer` instead of `Response.Redirect`.
- Do not have long-running tasks.
- Avoid lock contention or blocking threads for any reason.

ADO.NET

- Store connection, command, parameter, and other database-related objects in reused fields rather than re-instantiating them on each call of a frequently invoked method.
- Pool network connections.
- Ensure database is properly structured and indexed.
- Reduce round-trip queries to the backend database.
- Cache whatever data you can locally, in memory.
- Use stored procedures wherever possible.
- Use paging for large data sets (i.e., do not return the entire data set at once).
- Batch requests if possible.
- Use `DataView` objects on top of `DataSet` objects rather than re-querying the same information.
- Use `DataReader` if you can live with a short-lived, forward-only iterator view of the data.
- Profile query performance using SQL Query Analyzer.

WPF

- Use the latest version of .NET – significant performance improvements have happened through the years.
- Never do significant processing on the UI thread.
- Ensure there are not any binding errors.

- Have only the visuals you absolutely need. Too many transformations and layers will slow down rendering.
- Reduce the size and depth of the visual tree.
- Use the smallest animation frame rate you can get away with.
- Use virtual views and lists to render only visible objects.
- Consider deferred scrolling for long lists, if necessary.
- `StreamGeometry` is faster than `PathGeometry`, but supports fewer features.
- `Drawing` objects are faster than `Shape` objects, but support fewer features.
- Update render transformations instead of replacing them, where possible.
- Explicitly force WPF to load images to the size you want, if they will be displayed smaller than full-size.
- Remove event handlers from objects to ensure they get garbage collected.
- Override `DependencyProperty` metadata to customize when changing values will cause a re-render.
- Freeze objects when you want to avoid change notification overhead.
- Prefer static resources over dynamic resources.
- Bind to CLR objects with few properties, or create wrapper objects to expose a minimal set of properties.
- Disable hit testing for large 3-D objects if not needed.
- Recompile for Universal Windows Platform, targeting Windows 10 to achieve significant performance improvements for free.

Appendix C

Big O Notation

At a layer above direct performance profiling lies algorithmic analysis. This is usually done in terms of abstract operations, relative to the size of the problem. Computer science has a standard way of referring to the cost of algorithms, called “Big O” analysis.

Big O

Big O notation, also known as asymptotic notation, is a way of summarizing the performance of algorithms based on problem size. The problem size is usually designated n . The “Big O”-ness of an algorithm is often referred to as its complexity. The term asymptotic is used as it describes the behavior of a function as its input size approaches infinity.

As an example, consider an unsorted array that contains a value we need to search for. Because it is unsorted, we will have to search every element until we find the value we are looking for. If the array is of size n , we will need to search, worst case, n elements. We say, therefore, that this linear search algorithm has a complexity of $O(n)$.

That is the worst case. On average, however, the algorithm will need to look at $n/2$ elements. We could be more accurate and say the algorithm is, on average, $O(n/2)$, but this is not actually a significant change as far as the growth factor (n) is concerned. Constants are dropped, leaving us with the same $O(n)$ complexity.

Big O notation is expressed in terms of functions of n , where n is the input size, which is determined by the algorithm and the data structure it operates on. For a collection, it could be the number of items in the collection; for a string search algorithm, it is the length of the respective strings.

Big O notation is concerned with how the time required to perform an algorithm grows with ever-larger input sizes. With our array example, we expect that if the array were to double in length, then the time required to search the array would also double. This implies that the algorithm has linear performance characteristics.

An algorithm with complexity of $O(n^2)$ would exhibit worse than linear performance. If the input doubles, the time is quadrupled. If problem size increases by a factor of 8, then the time increases by a factor of 64, always squared. This type of algorithm exhibits quadratic complexity. A good example of this is the bubble sort algorithm. (In fact, most naive sorting algorithms have $O(n^2)$ complexity.)

```
private static void BubbleSort(int[] array)
{
    bool swapped;
    do
    {
        swapped = false;
        for (int i = 1; i < array.Length; i++)
        {
            if (array[i - 1] > array[i])
            {
                int temp = array[i - 1];
                array[i - 1] = array[i];
                array[i] = temp;
                swapped = true;
            }
        }
    }
```

```

    } while (swapped);
}

```

Any time you see nested loops, it is quite likely the algorithm is going to be quadratic or polynomial (if not worse). In bubble sort's case, the outer loop can run up to n times while the inner loop examines up to n elements on each iteration, therefore the complexity is $O(n^2)$.

When analyzing your own algorithms, you may come up with a formula that contains multiple factors, as in $O(8n^2 + n + C)$ (a quadratic portion multiplied by 8, a linear portion, and a constant time portion). For the purposes of Big O notation, only the most significant factor is kept and multiplicative constants are ignored. This algorithm would be regarded as $O(n^2)$. Remember, too, that Big O notation is concerned with the growth of the time as the problem size approaches infinity. Even though $8n^2$ is 8 times larger than n^2 , it is not very relevant compared to the growth of the n^2 factor, which far outstrips every other factor for large values of n . Conversely, if n is small, the difference between $O(n \log n)$, $O(n^2)$, or $O(2n)$ is trivial and uninteresting. Note that, you can have complex significant factors such as $O(n^2 2^n)$; and neither component involving n is removed (unless it really is trivial).

Many algorithms have multiple inputs and their complexity can be denoted with multiple variables, e.g., $O(mn)$ or $O(m + n)$. Many graph algorithms, for example, depend on the number of edges and the number of vertices.

The most common types of complexity are:

- $O(1)$ (Constant): The time required does not depend on size of the input. Many hash tables have $O(1)$ complexity.
- $O(\log n)$ (Logarithmic): Time increases as a fraction of the input size. Any algorithm that cuts its problem's space in half on each iteration exhibits logarithmic complexity. Note that there is no specific base for this log.
- $O(n)$ (Linear): Time increases in proportion with input size.

- $O(n \log n)$ (Loglinear): Time increases quasilinearly, that is, the time is dominated by a linear factor, but this is multiplied by a fraction of the input size.
- $O(n^2)$ (Quadratic): Time increases with the square of the input size.
- $O(n^C)$ (Polynomial): C is greater than or equal to 2.
- $O(C^n)$ (Exponential): C is greater than 1.
- $O(n!)$ (Factorial): Try every permutation.

Algorithmic complexity is usually described in terms of its average and worst-case performance. Best-case performance is not very interesting because, for many algorithms, luck can be involved (e.g., it does not really matter for our analysis that the best-case performance of linear search is $O(1)$ because that means it just happened to get lucky).

The following graph shows how fast time can grow based on problem size. Note that the difference between $O(1)$ and $O(\log n)$ is almost indistinguishable even out to relatively large problem sizes. An algorithm with $O(n!)$ complexity is almost unusable with anything but the smallest problem sizes.

Though time is the most common dimension of complexity, space (memory usage) can also be analyzed with the same methodology. For example, most sorting algorithms are $O(\log n)$ in time, but are $O(n)$ in space. Few data structures use more space, complexity-wise, than the number of elements in the data structure.

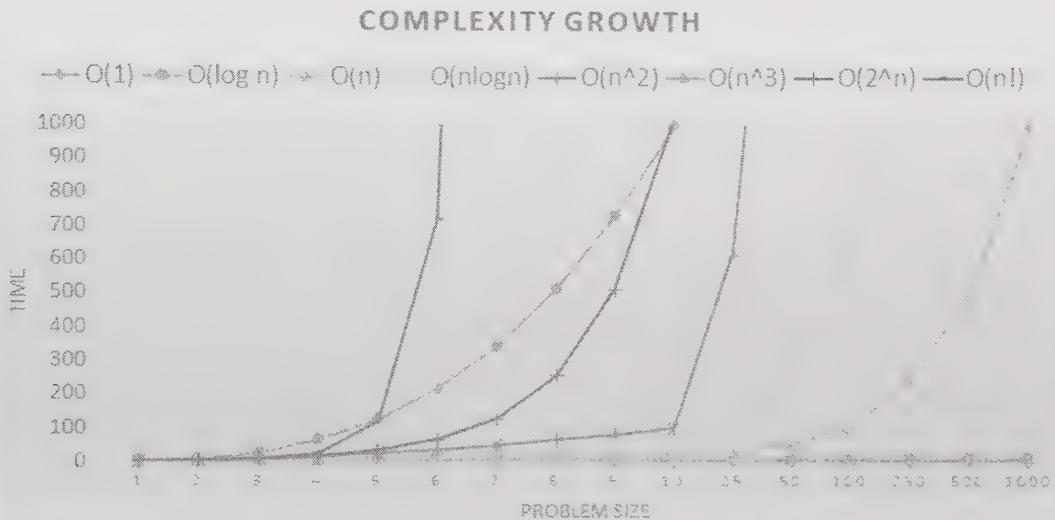


Figure C.1. Problem size vs. growth rate for various types of algorithms.

Common Algorithms and Their Complexity

Sorting

- Quicksort: $O(n \log n)$, $O(n^2)$ worst case
- Merge sort: $O(n \log n)$
- Heap sort: $O(n \log n)$
- Bubble sort: $O(n^2)$
- Insertion sort: $O(n^2)$
- Selection sort: $O(n^2)$

Graphs

- Depth-first search: $O(E + V)$ (E = Edges, V = Vertices)

- Breadth-first search: $O(E + V)$
- Shortest-path (using Min-heap): $O((E + V) \log V)$

Searching

- Unsorted array: $O(n)$
- Sorted array with binary search: $O(\log n)$
- Binary search tree: $O(\log n)$
- Hash table: $O(1)$

Special Case

- Computing every permutation of a string: $O(n!)$
- Traveling salesman: $O(n!)$ (Worst-case. There is actually a way to solve this in $O(n^2 2^n)$ using dynamic programming techniques.)

Often, $O(n!)$ is really just shorthand for “brute force, try every possibility.”

Appendix D

Bibliography

Useful Resources

- Hewardt, Mario and Patrick Dussud. *Advanced .NET Debugging*. Addison-Wesley Professional, November 2009.
- Richter, Jeffrey. *CLR via C#*, 4th ed. Microsoft Press, November 2012.
- Russinovich, Mark and David Solomon, Alex Ionescu. *Windows Internals*, 6th ed. Microsoft Press, March 2012.
- Rasmussen, Brian. *High-Performance Windows Store Apps*. Microsoft Press, May 2014.
- *ECMA C# and CLI Standards*: <https://www.visualstudio.com/license-terms/ecma-c-common-language-infrastructure-standards/>, Microsoft, retrieved 23 November 2017.
- *Amdahl's Law*, <http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf>

People and Blogs

In addition to the resources mentioned above, there are a number of useful people to follow, whether they write on their own blog or in articles for various publications.

- .NET Framework Blog: Announcements, news, discussion, and in-depth articles. <http://blogs.msdn.com/b/dotnet/>.
- Maomi Stephens: CLR developer and GC expert. Her blog at <http://blogs.msdn.com/b/maoni/> is updated infrequently, but there is a lot of useful information there and important announcements occasionally show up.
- Vance Morrison: .NET Performance Architect. Author of PerfView tool, MeasureIt, and numerous articles and presentations on .NET performance. Blogs at <http://blogs.msdn.com/b/vancem/>.
- Matt Warren: .NET performance enthusiast, Microsoft MVP, blogger, and contributor to many .NET open source projects, including BenchmarkDotNet. <http://mattwarren.org/>
- Brendan Gregg: <http://www.brendangregg.com/>. Not a .NET guy, but there is a ton of useful performance information here.
- MSDN Magazine: <http://msdn.microsoft.com/magazine>. There are lots of great articles going into depth about CLR internals.

Contact Information

Ben Watson

Email: feedback@writinghighperf.net

Website: <http://www.writinghighperf.net>

Blog: <http://www.philosopicalgeek.com>

LinkedIn: <https://www.linkedin.com/in/benmwatson>

Twitter: <https://twitter.com/benmwatson>

If you find any technical, grammatical, or typographical errors please let me know via email and I will correct them for future versions.

If you wish to purchase an electronic edition of this book for your organization, please contact me for license information.

If you enjoyed this book, please leave a review at your favorite online retailer.
Thank you!

Index

- .NET Compiler Code Analyzers, 417, 425–436
- .NET Core, xxviii, xli
 - LINQ, 375
- .NET Framework
 - .NET Core, xli
 - source code, xxxiv
 - all-purpose, 337, 365, 379
 - official blog, 466
 - pooling, 105
 - SOS.dll, 43
 - source code, 54
 - timeline, xxxvii
 - TPL, 222
- .NET Native, 196–197
- abstract base class, 295
- abstraction, 285, 338, 436
- `ActionBlock<T>`, 227
- `Activator.CreateInstance`, 321, 328
- ADO.NET, 457
- `AggregateException`, 219, 221–222, 226
- agility, xxxi
- algorithms, xxxiv, 460
 - complexity, 459
- Amdahl’s Law, 4, 439
- `AnalyzeFieldSymbol`, 428
- animation, 458
- `AppDomain.UnhandledException`, 224
- architecture, xxiv, xxv, xxx, xxxiv, 3, 455
 - processor and memory models, 259
- `ArrayList`, 340
- arrays, 341
 - jagged vs. multi-dimensional, 342–345
- large object heap, 73
- processing with SIMD, 380
- `ArraySegment<T>`, 98, 110, 341
- as, 307
- ASP.NET, 455–456
 - .NET Core, xli
- `async` and `await`, 244–248
 - ref-return, 303
- automation, 64, 442
- `AutoResetEvent`, 272
- availability, 6
- average, 5

- BatchBlock**, 227
benchmarking, 7–8, 56, 60, 442, 447
 BenchmarkDotNet, 57–60
Big O notation, 459–462
BitArray, 350
BitVector32, 350
Boolean logic, 351
bounds checking, xxvii
boxing, 55, 296–298
 collections, 340, 346
 finding, 330–333
BroadcastBlock<T>, 227
BufferBlock<T>, 227
- caching
 ADO.NET, 457
 ASP.NET, 456
 casting, 307
 delegates, 311, 324
 I/O, 376
 object resurrection, 125
 ToString, 359, 365
 weak references, 117
- CallSite**, 319
callvirt, 327
CancellationToken, 218, 226, 253
casting, 306–308, 346
 collections, 340
class, *see* reference type
clock interval, 210
ClockRes, 61
CLR
 internal details, xxxv
 timeline, xxxvii
CLR MD, 49–54
 boxing, 333
- enumerate heap objects, 51
finalizable objects, 180
fragmentation, 170
heap objects, 146
large objects, 163
method size, 202
object generation, 173
object roots, 156
object size, 161
survival, 174
- CLR Profiler**, 37
 allocations, 143
 fragmentation, 170
 heap analysis, 151
- code generation, 321–328
 JIT, 191
- code ownership, 446
code quality, xxiii
code reviews, 445–446
coding standards, 445
Cogswell, Bryce, 60
collections, 340–354
 .NET Core, xliv
 concurrent, 272–275, 348 350,
 354
 custom, 354
 generic, 345–348
 initial capacity, 351
 key comparison, 352–353
 lock-free, 266
 obsolete, 340
 pooling, 105
 sorting, 353
 synchronization, 264, 273
- ConcurrentBag<T>**, 272, 348

- ConcurrentDictionary<TKey, TValue>**, 272, 273, 348
ConcurrentQueue<T>, 272, 348
ConcurrentStack<T>, 272, 348
 contention
 ASP.NET, 456
 concurrent collections, 273
 double-checked locking, 261
 ETW events, 26, 278
 memory allocation, xxx, 68, 70
 performance counters, 278
 PerfView, 32, 282
SpinLock, 271
 thread scheduling, 210, 212
 timer queue, 251
 Visual Studio, 16
CoreInfo, 61
 correctness, 415
 costs and benefits, xxxi
CounterCreationData, 386
CreateProcess, 309
- data structures
 memory layout, xxix
 databases, 63
 indexing, 457
 paging, 457
DataReader, 457
DataSet, 457
DataView, 457
DateTime, 60, 366
DateTime.Parse, 361
 debugging, xxvii, xxxi, 43, 60, 65,
 91, 92, 108, 277, 416
 delegates, 311–314
 code generation, 324
- concurrent collections, 349
 lambdas, 313
 lazy initialization, 362
LINQ, 374
 parallel loops, 235
 tasks, 214
DependencyProperty, 458
DiagnosticSeverityInfo, 428
Dictionary<TKey, TValue>, 273,
 345, 352, 365
 foreach, xxxvi
IDictionary<TKey, TValue>, 354
 direct memory access, 199
Diskmon, 61
DiskView, 61
Dispose, 94, 102
dotPeek, 54
dynamic, 190, 316–319
 dynamic code
 .NET Native, 197
 Dynamic Language Runtime, 316
DynamicMethod, 323
- ego, 446
Enum.HasFlag, 364
Enum.isDefined, 365
Enum.ToString, 365
 enumerate heap objects
 CLR MD, 51
 enums, 364–366
 Event Tracing for Windows, 12, 26,
 60, 393–394
 defining events, 394–399
 event levels, 397
 exceptions, 330
 hardware events, 40

JIT events, 201
keywords, 397
listening for events, 401–406
manifest, 407
metadata, 407–409
PerfView, 30, 399
reflection, 398
thread and contention events,
 278
Windows Performance Analyzer,
 40

`EventListener`, 401–406
`EventSource`, 394–399
`Exception.StackTrace`, 316
`Exception.ToString`, 221
exceptions, 314–316
.NET Framework APIs, 361
catching first-chance, 333
ETW events, 330
logging, 221
unhandled, 221

`File`, 375
`FileOptions`, 375
`FileStream`, 375
`FileSystemWatcher`, 165
finalizers, 70, 129, 280
 diagnosing, 179–180
ETW events, 131
exceptions, 221
freachable, 180
garbage collection, 94–97
pooling, 105
resurrection, 125, 179

`fixed`, 93, 164
`Flags`, 365

`foreach`, xxxvi, 57, 303–306
`FormatException`, 360, 361
fragmentation
 see memory, fragmentation, xxx

`Func<T>`, 275, 362
`FxCop`, 417
 command line, 423
 custom rules, 417–425
 SDK, 418

garbage collection
 .NET Core, xli
 allocation, 68, 72, 275
 allocation profiling, 140–144
 allocation tick, 140
 automatic tuning, 78
 background, 80–81
 card table, 75
 compaction, 72, 82, 97, 451
 configuration, 78–88
 `gcConcurrent`, 81
 `gcServer`, 79
 `COMPLUS_GCHeapCount`, 86
 `COMPLUS_GCNоАffinitize`, 87
 `COMPLUS_HeapVerify`, 88
 `gcAllowVeryLargeObjects`,
 85
 latency modes, 81–84
 cross-generation references, 93
 determining roots, 155–158
 deterministic, xxxii
 ETW events, 130–132
 finalizers, 94–97, 129
 fragmentation, 97
 GC handles, 73, 127
 generations, 71, 128

- induced, 111–112, 178
large object heap, 71, 73, 90, 97,
 129, 361, 450
 discovering objects, 162
LINQ, 374
load-balancing, 92
low-latency, 82, 111
memory management, 67
most important rule, 89
multiple heaps, 79
notifications, 113–117, 451
object generation, 173
object lifetime, 91, 451
on-demand compaction, 74
operation, 71–76
overhead, 67
parallelism, 92
pause time, 136, 451
performance counters, 127
phases, 75
pinning, 128, 164–166
pooling, 90, 102–106, 450
promotion, 90, 91
roots, 73
segments, 71–73, 79, 176
 no-GC regions, 83
server GC, 70, 79, 450
small object heap, 71
survival, 174
weak references, 117
workstation GC, 78, 450
`GC.Collect`, 82, 112, 113, 178
`GC.GetGeneration`, 173
`GC.RegisterForFullGCNotification`,
 114
`GC.ReRegisterForFinalizer`, 125
`GC.SuppressFinalize`, 94
`GC.TryStartNoGCRegion`, 83–84
`GC.WaitForFullGCApproach`, 114
`GC.WaitForFullGCComplete`, 114
`GCCollectionMode`, 112
`GCHandleType.Pinned`, 93
`GCLargeObjectHeapCompactionMode`,
 113
`GCSettings.LatencyMode`, 81
`GetBuffer`, 109
goals, 2, 449

Handle (tool), 61
hardware, xxvi, 416
hardware interface, xxix
`HashSet<T>`, 345
`Hashtable`, 340
heap
 analysis, 48, 144–151
 dump
 CLR MD, 53
 CLR Profiler, 38
 ProcDump, 61
 WinDbg, 47
 layout, 71–75
 visualization, 132–135
`HttpClient`, 361
`HttpWebRequest`, 361
`HybridDictionary`, 340

I/O
 asynchronous, 238–244, 452
 blocking, 282
 buffer size, 243
 connection pooling, 456, 457
 HTTP, 377–379, 436
 compression, 456

KeepAlive, 378
WinHTTP, 379
random access, 376
reading files, 375–377
IAsyncResult, 237
ICollection<T>, 105, 354
IComparable<T>, 293, 353
IDisposable, 94, 239
 pooling, 102
IEnumerable, 57
IEnumerable<T>, 234, 305, 354, 372, 374
IEquatable<T>, 291
IL analyzers, 54
ILDASM, 330
 boxing, 296
IList<T>, 105, 354
ILSpy, 54
infrastructure, 442
inlining, 185, 189
 NGEN, 194
Int32.Parse, 360, 361
Int32.TryParse, 360
interface dispatch, 294–295
Interlocked, 258, 259, 273, 348, 363, 452
Interlocked.Add, 266
Interlocked.CompareExchange, 265, 266
Interlocked.Decrement, 266
Interlocked.Exchange, 266
Interlocked.Increment, 265, 266
InvalidCastException, 306
IPAddress, 359
is, 307
ISourceBlock<TOutput>, 227
ITargetBlock<TInput>, 227
JIT, xxix, 183–184
 .NET Core, xli
 advantages, 184
 benchmarks, 8
 custom warmup, 198–199
 dynamic, 319
 ETW Events, 201
 generics, 184
 IL size, 202
 inlining, 189, 290
 instruction reordering, 259
 locality, xxx
 multicore JIT, 192
 optimizations, 289–290, 327, 452
 performance, 205
 performance counters, 200
 Profile Guided Optimization, 452
 range-check elimination, 189
 regular expressions, 368
 startup time, 452
 ThePreStub, 188
 warmup, 184
Knuth, Donald, 4
lazy initialization, 362–363
Lazy<T>, 261, 362
LazyInitializer.EnsureInitialized, 363
LazyThreadSafetyMode, 363
LinkedList<T>, 345
LinkedListNode<T>, 347
LINQ, 352, 370–375
 .NET Core, xlii

JIT, 190
 Parallel, 374
List<T>, xlvi, 345
 foreach, xxxvi
ListDictionary, 340
ListDLLs, 61
lock, *see* Monitor
log4net, 393
logman, 27

 maintainability, 437, 440
Managed Profile Guided Optimization (MGPO), 195–196
ManualResetEvent, 272
ManualResetEventSlim, 272
MarshalByRefObject, 263
MeasureIt, 56, 60, 303, 309
memory, *see also* garbage collection
 address space, 173
 allocation, xxx
 contention, 70
 finalizer, 70
 rate, 88
 allocation buffer, 68, 70
 corruption, 436
 direct access, 298
 fragmentation, xxx, 67, 68, 111,
 113, 151, 166–173, 451
 large object heap, 110
 virtual memory, 171–173
 JIT efficiency, 185
 leak, 151
 pooling, 106
 locality, 68, 340, 346, 354
 low-fragmentation heap, 68
 native heap, 68

 object size, 158–162
 paging, 2
 types, 2, 18
 memory model, 258–259
MemoryStream, 98, 106
MemoryStream.GetBuffer, 308
MemoryStream.Length, 308
MemoryStream.ToArray, 110
MethodImplOptions.Synchronized,
 271
MethodInfo, 320, 326
MethodInfo.Invoke, 321, 328
metrics, 2, 449
 tracking, 63
Microsoft.Diagnostics.Runtime, *see*
 CLR MD
Monitor, 56, 257, 259, 261–263, 271,
 273, 348
Monitor.TryEnter, 263
monitoring, 9
Morrison, Vance, 29, 56, 295, 466
mscordacwks.dll, 53
MSIL, xxix, 183
Mutex, 272

NameValueCollection, 341
NGEN, 193–196, 452
NoInlining, 186
NTFSInfo, 61
NuGet, xlivi
NullReferenceException, 251

Object.Equals, 291, 292
Object.GetHashCode, 291, 292
Object.ToString, 359
operator!=, 292
operator==, 292

optimization, xxvi
 premature, 4

`OrderedDictionary`, 341

`OutOfMemoryException`, 173

overhead, xxiii, xxx, 64

`P/Invoke`, 199, 308–311, 362
 .NET Core, xlvi
 performance counters, 329
 security checks, 310

page fault, 21

`Page.DataBind`, 456

`Page.IsPostBack`, 456

Parallel LINQ, 374

`Parallel.For`, 234, 450

`Parallel.ForEach`, 234

`ParallelLoopState`, 234

`Partitioner`, 235

`PathGeometry`, 458

percentile, 5

`PerfMon`, 18
 alert, 23

performance counters, 18, 385–386
 .NET, 382
 averages, 387
 deltas, 389
 garbage collection, 127
 installing, 386–387
 instantaneous, 389
 JIT, 200
 percentages, 390

`PerformanceCounter`, 386

`PerformanceCounterInstaller`,
 387

`PerformanceCounterPermission`,
 387

PerfView, 29, 450
 allocations, 450
 boxing, 332
 concurrency, 282
 custom extension, 411–413
 ETW, 394, 399
 exceptions, 334
 folding patterns, 32
 fragmentation, 170
 GC performance, 136
 grouping and folding, 32, 382
 heap analysis, 149
 heap diffs, 153
 I/O blocking, 282
 JIT, 205, 375, 452
 large object allocations, 163
 manual, 37
 object roots, 152
 object size, 162
 pinned references, 166
 threading analysis, 452

pinning
 garbage collection, 93–94, 152
 P/Invoke, 309

pointers, xxvii, xxxi, 126, 298, 309
 arithmetic, 99
 dereferencing, 285, 341
 managed buffers, 199
 method table, 286
 size, 184, 285
 stack traces, 316

polymorphism
 interfaces, 295

preprocessing, 329

`ProcDump`, 61

Process Explorer, 61

- Process Monitor, 61
`Process.GetProcesses`, 361
processor affinity, 80
 garbage collection, 85
`ProcessorAffinity` property, 86
productivity, xxvii, 339
`ProfileOptimization.StartProfile`,
 193
profiling, xxxiii, xxxix, 12, 459
 command line, 16–17
 concurrency, 12
 CPU, 450
 limitations, 31
 ETW events, 26, 394
 memory, 38, 140, 361
 overhead, 64
 Visual Studio, 11
properties, 190, 290
`PsInfo`, 61
quantifiability, 444–445
`Queue`, 340
`Queue<T>`, xlvi, 345
race conditions
 timers, 250
`RAMMap`, 61
`Random`, 275
readability, 437
`ReaderWriterLock`, 56, 272
`ReaderWriterLockSlim`, 272
`ReadOnlySpan<T>`, 101, 360
`RecyclableMemoryStream`, 106–110
`RecyclableMemoryStreamManager`,
 108
ref-return, 287, 298–303
reference types
dereferencing, 287
overhead, 286
thread safety, 293
reflection, 319–321
 .NET Native, 197
 `Equals`, 291
 ETW events, 398
Reflector, 54
`Regex`, 368
`RegexOptions.Compiled`, 369
regular expressions, 368–370
 .NET Core, xlvi
 code generation, 328
 JIT, 190
 timeouts, 370
reliability, 415
requirements, xxiv, 1
resource constraints, 449
`Response.Redirect`, 456
responsiveness, 3
resurrection, 124–125
return on investment, 443
Russinovich, Mark, 60
`RyuJIT`, 190
scalability, 440
`SDelete`, 61
`sealed`, 290
security, xxvii, 440
`Semaphore`, 272
`SemaphoreSlim`, 268, 272
serialization
 .NET Native, 197
 XML, 328, 339
`Server.Transfer`, 456
`ServicePoint`, 378

ServicePointManager, 377
SIMD, xxviii, 200, 380–382
simplicity, xxxv
`SortedDictionary< TKey, TValue >`, 345
`SortedList`, 340
`SortedList< TKey, TValue >`, 345
`SortedSet< T >`, xlvi, 345
SOS, 43
`Span< T >`, 98, 341
`SpinLock`, 271
SQL Query Analyzer, 457
stability, xxxi, 415
`Stack`, 340
`Stack< T >`, 345
`stackalloc`, 100, 126 127
`StackOverflowException`, 127
static code analysis, 416
statistics, 5, 449
 benchmarking, 7
 garbage collection, 136
`Stopwatch`, 60, 366
`Stopwatch.Frequency`, 367
`Stopwatch.GetTimestamp`, 367
stored procedures, 457
`Stream`, 199, 238
`Stream.BeginRead`, 243
`Stream.EndRead`, 243
`StreamGeometry`, 458
`String`, 354–361
 comparisons, 355–356
 concatenation, 356
 encoding, 379
 .NET Core, xlvi
 immutability, 354
 large object heap, 73
 parsing, 360
 substrings, 360–361
`string`
 as synchronization objects, 263
`String.Compare`, 355–356
`String.Concat`, 356, 359
`String.Equals`, 356
`String.Format`, 358–359
 boxing, 296
`String.Join`, 356
`String.ToLower`, 352, 356, 425, 431
`String.ToUpper`, 352, 356, 425, 431
`StringBuilder`, 356, 359
 initial capacity, 358
`StringCollection`, 341
`StringComparer.OrdinalIgnoreCase`, 353
`StringComparison.CurrentCulture`, 355
`StringComparison.Ordinal`, 355
`StringComparison.OrdinalIgnoreCase`, 355
`StringDictionary`, 341
Strings (tool), 61
struct, *see* value types
`SuppressUnmanagedCodeSecurity`, 310
symbol server, 49
`SyntaxFactory`, 431
`SysInternals`, 60
`Task`, 214–226, 416, 450
 cancellation, 217, 226
 child tasks, 224–226
 continuations, 214–217
 exceptions, 219–221, 225

unhandled exceptions, 221
Task Parallel Library, 214
`Task.ContinueWith`, 214
`Task.Delay`, 251
`Task.Result`, 219
`Task.Run`, 225
`Task.Start`, 214
`Task.StartNew`, 225
`Task.Wait`, 219, 237, 247
`TaskCanceledException`, 226
`TaskCompletionSource<T>`, 239, 246
`TaskCompletionSource<T>.TrySetResult`, 239
`TaskContinuationOptions`, 216, 217, 248
`TaskCreationOptions`, 216, 225
`TaskFactory.FromAsync`, 237, 243
teams, 439
testing, 440–441
thread-local data, 275–276
`Thread.Abort`, 253
`Thread.Sleep`, 210, 249, 250
`ThreadLocal<T>`, 275
`ThreadPool.SetMaxThreads`, 252
`ThreadPool.SetMinThreads`, 252
threads, 209–211
 blocking, 237
 classes and thread safety, 293
 concurrency
 profiling, 12, 17
 contention, 278, 452, 456
 ETW Events, 278
 parallel loops, 233–236
 delegates, 235
 performance counters, 277–278
priority, 213, 253–254
program structure, 247–249
quantum, 210, 213
scheduling, 210, 416
synchronization, 254–276
asynchronous locks, 268–270
collections, 272–275
deadlock, 263
double-checked locking, 260–261
scope, 264
synchronization objects, 263
thread-local data, 275–276
thread pool, 211–214, 453
 automatic tuning, 252
 timer queue, 250
UI thread, 457
`ThreadStatic`, 276
time
 code execution, 60
 measuring, 366–367
`Timer`, 249
timers, 249–252
 race condition, 250
 tick resolution, 250
`TimeSpan`, 366
total cost of ownership, 443
Toub, Stephen, 236, 270
TPL Dataflow, 226–233
`TraceEvent`, 30, 409
training, 446–447
`TransformBlock<TInput, TOutput>`, 227
`TransformManyBlock<TInput, TOutput>`, 227
`Tuple`, 235, 293

Type, 326
type loading, 192
.NET Native, 197
dynamic, 194, 321
type safety, xxvii
UI transformations, 458
Universal Windows Platform applications, 196
Universal Windows Platform, 458
unmanaged code
risk, 436–437
`UnmanagedMemoryStream`, 199
using
 `async` and `await`, 246
value types
 access efficiency, 287
 blittable, 291, 309
 boxing, 296, 322
 collection, 293
 interfaces, 296
 locality, 287
 maximum size, 286
 overhead, 286
`ValueTuple`, 293
`ValueType.Equals`, 291
`Vector`, 381
`ViewState`, 456
virtual, 289
`VirtualAlloc`, 71
Visual Studio, 10
 allocations, 140
 Concurrency Visualizer, 12, 280
 Extensibility Tools, 425
 FxCop, *see* FxCop
 heap analysis, 148
instrumentation, 16
IntelliSense, xxxv
Memory Usage profiler, 159
profiling, 11, 12
Profiling Wizard, 280
sample projects, xliii
Standalone Profiler, 16, 450
Syntax Visualizer, 434
`VMMMap`, 61, 135
 fragmentation, 171
`volatile`, 259, 261, 274
weak references, 117–118, 178
`WeakReference<T>`, 117, 119, 124
WinDbg, 43–49
 commands
 `address`, 172
 `bp`, 177
 `bpmd`, 178
 `dump`, 47
 `DumpHeap`, 48, 145, 146, 155,
 168, 175, 180
 `DumpObj`, 146, 160
 `DumpStack`, 178
 `DumpStackObjects`, 48, 155
 `eeheap`, 76, 132, 169, 176
 `FinalizeQueue`, 179
 `FindRoots`, 175
 `FindRoots`, 156
 `g`, 46
 `gchandles`, 165, 178
 `gcroot`, 155
 `gcwhere`, 173
 `HeapStat`, 133, 167
 `kb`, 48
 `loadby`, 46

ObjSize, 160
ProcInfo, 46
s, 48
sx, 334
sxe, 46
symfix, 49
Threads, 279
ThreadState, 279
U, 206
VerifyHeap, 87
VMMap, 133
Windows 10, 458
Windows Forms
 and .NET Core, xli
Windows Performance Analyzer, 40
WPF, 457–458
 .NET Core, xli
WriteOnceBlock<T>, 227
Xamarin, xxviii
XML, 339



32526048R00293

Made in the USA
San Bernardino, CA
15 April 2019

WRITING HIGH-PERFORMANCE .NET CODE

2ND
EDITION

Take performance to the next level!

This book does not just teach you how the CLR works—it teaches you exactly what you need to do now to obtain the best performance today. It will expertly guide you through the nuts and bolts of extreme performance optimization in .NET, complete with in-depth examinations of CLR functionality, free tool recommendations and tutorials, useful anecdotes, and step-by-step guides to measure and improve performance.

This second edition incorporates the advances and improvements in .NET over the last few years, as well as greatly expanded coverage of tools, more topics, more tutorials, more tips, and improvements throughout the entire book.

Ben Watson has been a software engineer at Microsoft since 2008. On the Bing platform team, he has built one of the world's leading .NET-based, high-performance server applications, handling high-volume, low-latency requests across thousands of machines for millions of customers.

In his spare time, he enjoys books, music, the outdoors, and spending time with his wife Leticia and children Emma and Matthew. They live near Seattle, Washington, USA.

<http://www.writinghighperf.net>

<http://www.facebook.com/writinghighperf.net>

<http://www.linkedin.com/in/benmwatson/>

<http://www.twitter.com/benmwatson>

New in the 2nd Edition:

- 50% increase in content!
- New examples, code samples, and diagrams throughout entire book.
- More ways to analyze the heap and find memory problems.
- More tool coverage, including expanded usage of Visual Studio.
- More benchmarking.
- New GC configuration options.
- Code warmup techniques.
- New .NET features such as ref-returns, value tuples, SIMD, and more.
- More detailed analysis of LINQ.
- Tips for high-level feature areas such as ASP.NET, ADO.NET, and WPF.

Also find expanded coverage and discover new tips and tricks for:

- Profiling with multiple tools to quickly find problem areas.
- Detailed description of the garbage collector, how to optimize code for it, and how to diagnose difficult memory-related issues.
- How to analyze JIT and warmup problems.
- Effective use of the Task Parallel Library to maximize thread utilization.
- Which .NET features are safe to use and which to avoid.
- Instrumenting your program with performance counters and events.
- Use the latest and greatest .NET features.
- Building a performance-minded team.
- ...and so much more!

T3-AFF-761



ISBN 9780990583455

A standard 1D barcode representing the ISBN number 9780990583455. To the right of the barcode is the number "90000 >".

9 780990 583455