Writing High-Performance .NET Code

2nd Edition

Ben Watson

Writing High-Performance .Net Code

Ben Watson

Writing High-Performance .NET Code

Version 2.1
PDF Edition
ISBN–13: 978–0–990–58346–2
ISBN–10: 0–990–58346–5

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.bluekittycreations.co.uk.

Contents

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

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 sugarcoat 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.

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:

$$\begin{aligned} &8 \text{ bytes array overhead} \\ &+ (4 \text{ byte pointer size} \times 1000000) \\ &+ (8 \text{ bytes overhead} + 16 \text{ bytes data }) \times 1000000 \\ &= 28 \text{ MB} \end{aligned}$$

For an array of structs, the results are dramatically different:

$$\begin{array}{r} 8 \text{ bytes array overhead} \\ +\,(16 \text{ bytes data} \times 1000000) \\ \hline = 16 \text{ MB} \end{array}$$

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.

Story

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 of 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.IEnumerator::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 issues 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 `int`s 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 JIT-ted.

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.

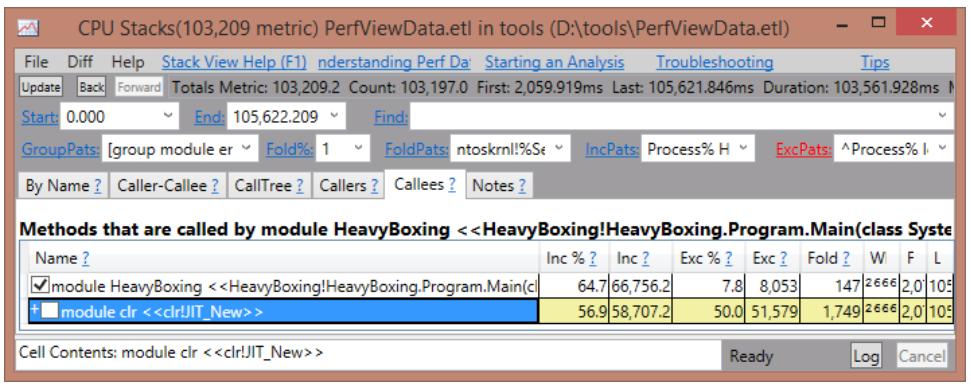

Figure 5.1. Boxing will show up in a CPU trace under the `JIT_New` method, which is the standard memory allocation method.

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.

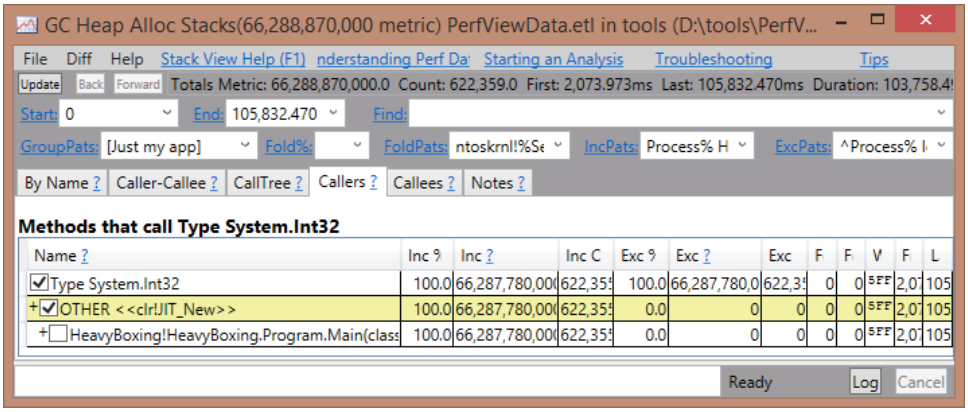

Figure 5.2. You can see in this trace that the ***Int32*** *is being allocated via* ***new****, which should not feel right.*

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.

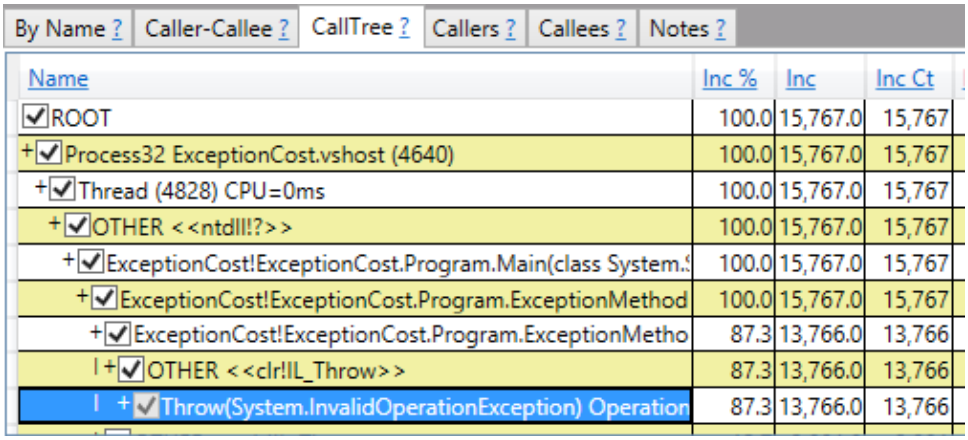

Figure 5.3. PerfView makes finding where exceptions are coming from trivially easy.

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.

Appendix D

Bibliography

Useful Resources

- Hewardt, Mario and Patrick Dussud. *Advanced .NET Debugging*. Addison-Wesley Professional, November 2009.

- Richter, Jeffrey. *CLR via C#, 4^{th} ed.* Microsoft Press, November 2012.

- Russinovich, Mark and David Solomon, Alex Ionescu. *Windows Internals, 6^{th} 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/.
- Maoni 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://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 lot 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.philosophicalgeek.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