Foreword

The original Desktop .NET Framework turned 10 years old recently (Feb 2012). I have been on the team since its very inception, and for over half that time I have acted as its performance architect, so that 10th birthday got me thinking about where .NET has been, where it is going, and what's the 'right' way to think about .NET performance. The chance to write a foreword on a book dedicated to .NET performance gave me the opportunity to write these thoughts down.

Programmer productivity has always been and will always be the fundamental value of the .NET framework. Garbage collection (GC) is the single most important feature that boosts productivity, not only because it avoids a broad class of nasty bugs (memory corruption), but also because it allows class libraries to be written without being "cluttered" with error-prone resource allocation conventions (no more passing buffers, or subtle rules about who is responsible for deleting memory). Strong type safety (which now includes Generics) is another important pillar because it captures a broad class of programmer intent (e.g., this list is homogeneous) and allows tools to find many bugs well before the program is ever run. It also enforces strong contracts between software components, which is very important for class libraries and large projects. The lack of strong typing in languages like JavaScript will always place them at a disadvantage as software scales. On top of these two pillars we added a class library that was designed for ease of use (very uniform, simple interfaces, consistent naming conventions, etc.). I am very proud of the result; we have built a system for building code whose productivity is second to none.

Programmer productivity alone is not enough, however, especially for a mature piece of software like the .NET runtime. We want high performance as well, and this is where this book comes into play.

The bad news is that in the same way that you can't expect your program to run correctly the very first time you run it, you can't expect high performance to "just happen" the very first time your program runs correctly. In the same way that there are tools (GC and type safety) and techniques (asserts, interface contracts) that reduce the occurrence of software bugs, there are tools (various profilers) and techniques (perf planning, hot path prototyping, lazy initialization) that reduce the likelihood of performance issues.

The good news is that performance follows the 90%-10% rule. Typically well over 90% of your application is *not* performance critical and can be written to maximum programmer productivity (the fewest, simplest, easiest lines of code possible). The other 10%, however, needs careful attention. It needs a plan, and it needs it *before* the code is written. This is what Chapter 1 is all about. To do that planning you need data (how fast are various operations, library calls, etc.), and for that you need measuring tools (profilers), which is what Chapter 2 is all about. These are the cornerstones of any high-performance software project. Heed these chapters well. If you take them to heart, you *will* write high-performance software.

The rest of the book is about the details you need to "know your options" when you are building your performance plan. There is no substitute for a basic understanding of the performance characteristics of the platform you are using, and that is what Chapter 3 (types), Chapter 4 (GC), and Chapter 5 (fundamental libraries) are about. If your prototyping shows that a simple, straightforward .NET implementation does not get you the performance you need, you should first investigate algorithm improvement (Chapter 9) or concurrency (Chapter 6) because these are likely to have the biggest effect. If you still need perf, some .NET specific tricks (Chapter 10) may help. If all else fails, you can sacrifice the programmer productivity for a small fraction of your code and write the most critical pieces in unsafe or unmanaged code (Chapter 8).

The key point I want to stress, however is having a plan (Chapter 1), because that is where it all starts. In this plan you will identify your high volume or performance critical paths and you will spend extra development time to carefully measure, and prototype solutions for these important paths. Next, armed with the measurements from the prototyping and the information in this book, it is typically a very straightforward exercise to get the performance you need. It might be as simple as avoiding common performance pitfalls, or maybe doing simple optimizations. It might be parallelizing your hot path, or maybe you have to write some unsafe code or unmanaged code. Whatever it is, the result is a design that achieves your performance objectives (by doing extra work on 10% of your code) while simultaneously reaping the productivity benefits of .NET on the remaining 90%. This is the goal of the .NET framework: high productivity *and* high performance. You *can* have it all.

So there you have it. The bad news is that performance is not free; you have to *plan* for it. The good news is that it is not that hard, and that by reading this book you have taken the most important step in writing your *high-performance* .NET application.

Good luck and enjoy the book. Sasha, Dima, and Ido did a great job on it.

Vance Morrison

Performance Architect, .NET Runtime