Maximizing .NET Performance

NICK WIENHOLT

Apress™

ISBN (pbk): 1-59059-141-0

Printed and bound in the United States of America 10987654321

Technical Reviewer: Simon Robinson

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Proofreader: Linda Seifert

Compositor: Gina Rexrode

Indexer: Michael Brinkman

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Foreword

"Managed code is too slow. We've got to do this in C++."

—Anonymous

Back in 1999, the ACM published a study[1] that presented a comparison of 40 independent implementations of a computationally intensive problem, created by different programmers in either Java—the then-current managed runtime environment—or C/C++. It concluded with the finding that interpersonal differences between the developers "are much larger than the average difference between Java and C/C++" and that "performance ratios of a factor of 30 or more are not uncommon between the median programs from the upper half versus the lower half."

This should teach you something: If you are not a guru-level C++ programmer, then the chance is quite high that a managed code implementation performs as well as the average C++ solution—especially given the fact that most .NET languages simply allow you fewer possibilities to introduce subtle memory-related or performance-related issues. And keep in mind that this study was conducted several years ago, and that Just-In-Time Compilation (JIT) as well as memory management and garbage collection (GC) technologies have been improved in the meantime!

This however doesn't mean that you can't create horribly slow, memory eating applications with .NET. That's why you should be really concerned about the other part of the study's conclusion, namely that "interpersonal differences . . . are much larger." In essence, this means that you have to know about how to optimize your applications so that they run with the expected performance in a managed environment. Even though .NET frees you from a lot of tasks that in C++ would have been your responsibility as a developer, these tasks still exist; these "little puppets" have only cleared the main stage and now live in some little corner behind the scenes. If you want your application to run in the top performance range, you will still need to find the right strings to pull to move these hidden figures and to basically keep them out of the way of negatively affecting your application's performance.

1. Lutz Prechtelt, "Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences," *Communications of the ACM* 42, no. 10 (October 1999): 109–112

But knowing about the common language runtime's internals is still not enough, as lots of performance issues actually turn up during application design and not just during the coding stage. Collections, remoting, interoperability with unmanaged code, and COM components are not the only things that come to my mind in this regard.

It is the aim of Nick's book to enable you to understand the design issues as well as the underlying CLR mechanisms in order to create the programs that run on the better side of the 30-times performance difference quoted in the ACM study. Nick really managed to create a book that addresses these issues, which will otherwise turn up when carelessly coding to a managed environment. This book will allow you to get into the details without being overwhelmed by the underlying complexity of the common language runtime.

The only thing you have to resist after reading the book is the urge to overoptimize your code. I was delighted to see that Nick begins with a discussion of identifying an application's performance-critical sections and only later turns towards isolating and resolving these *real* performance bottlenecks. This is, in my opinion, one of the most important tasks—and also one of the most complex ones—when working with large-scale applications.

And now read on and enjoy the ride to the better side of a 30-fold performance difference.

Ingo Rammer, author of *Advanced .NET Remoting*
Vienna, Austria
`http://www.ingorammer.com`

CHAPTER 1

Introduction

Software performance is a topic rife with paradoxes, confusion, and mystery. To provide a few examples:

- Processor speed doubles every 18 months, but performance concerns never disappear.

- Two competing software platforms both show order-of-magnitude speed advantages over the other in benchmarks produced by their respective vendors.

- Languages and technologies are shunned due to their perceived lack of performance, but a precise measurement of the slowness is rarely produced, and the criticism of perceived slowness is discussed even if performance is not a project priority.

This book attempts to peel back some of the mystery surrounding the performance of code that targets the .NET Framework. The book has two main goals—to present a detailed discussion on the performance of various .NET Framework technologies, and to illustrate practices for conducting reliable performance assessment on techniques and methodologies that are not covered in the book. To address both these goals, every benchmark discussed in this book is available for download from the publisher's Web site (`http://www.apress.com`), and each test is clearly numbered for quick location in the code samples. This allows for an easy reproduction of the test runs on the reader's own systems, and also for analysis of the code that makes up the test. Chapter 2 contains a detailed discussion on conducting performance assessments, and the benchmark harness used to conduct the benchmark results presented is fully documented in the appendix.

The reader is encouraged to critically analyze the results presented. Service packs and new versions of the .NET Framework and operating system will mean that some of the results presented in this book may differ from those the reader obtains. In other cases, small changes in a test case can significantly change the performance of a method, and a test case on a particular technology may not be relevant to other uses of the same technology. Subtle changes can bring significant performance improvements, and a keen eye for detail is critical.

Material Covered

The focus of this book is .NET Framework performance. The book concentrates on Framework performance from the ground up, and does not include discussions of higher-level technologies like Windows Forms, ASP.NET, and ADO.NET. These technologies are important, and future volumes may well be written that focus on them, but it is critical to appreciate that all .NET code is built on a common base, which is the focus of this book. System-level developers will get the most out of this book, but every attempt has been made to ensure that the material is accessible and relevant to application developers who use higher-level technologies. By developing a strong understanding of Framework performance, application developers are in a much better position to identify and avoid performance mistakes in both their own code and high-level application libraries.

Key areas of coverage include the following:

- Analyzing the performance of software systems using black-box and white-box assessment techniques
- Designing types (classes and structures) so they have optimum performance characteristics, and interact efficiently with Framework design patterns
- Using remoting to build high-performance distributed systems
- Interacting efficiently with unmanaged code using COM Interop, P/Invoke, and C++
- Understanding the interaction between performance and language selection
- Working with the .NET garbage collector to achieve high-performance object allocations and collections
- Selecting the correct collection class for maximum performance
- Locating and fixing performance problems

Comparison between .NET and competing platforms like J2EE is not covered. While this information may be of interest to some, cross-technology performance comparisons are prone to much controversy and conflicting results, as the recent Pet Shop war between Java vendors and Microsoft demonstrated. At the end of the day, most cross-technology comparisons are motivated by commercial considerations, and do not assist developers in writing faster and more efficient code on

either platform. This book is focused on giving developers the information and tools they need to write high-performance code using the .NET Framework, and cross-platform sniping would only detract from that goal.

For readers interested in independently verified benchmark comparisons between the offerings of various vendors, the Transaction Processing Performance Council results, available at `http://www.tpc.org`, are an excellent resource.

Solving Specific Performance Problems

If this book has been purchased to assist with optimization of a section of code that is taking a long time to complete its task, Chapter 15 is the optimal starting point. This chapter explores the tools and techniques available for determining the cause of poorly performing .NET applications, and will assist the reader in investigating and rectifying a performance problem, even if the specific answer to the problem is not contained in this book. The remaining chapters cover specific topics related to the .NET Framework and performance, and attempt to highlight relevant performance trade-offs, pitfalls, and optimizations for the particular technology.

Performance and the Development Process

A structured development process is crucial in producing software with adequate performance characteristics. Regardless of the particular development methodology that an organization follows, it is critical that the development priorities of the project or product reach the people who actually architect and develop the software system. In the absence of development priorities, a system will revert to a particular developer or architect's "natural state," and the emphasis may be on code readability and maintainability, development speed, performance, testability, or simply on technology familiarity. Whatever the case, in the absence of clear communication, it will only be through chance that developer and project priorities coincide, and such coincidental goal convergence is not likely to happen all that often.

Achieving the correct performance characteristics for a software application starts during the design phase of a project. If architects and business analysts fail to provide any guidance regarding performance targets, they should surrender their right to criticize the result of the development process as slow. Failing to define or communicate the priorities of a system to the developers and testers responsible for building the system invariably leads to some degree of disappointment with the finished result. If this book is being read with a view to correcting perceived performance problems that are the result of an inadequate

software development process, the reader is encouraged to also gain access to material on software development methodologies. Compensating for poor planning with post-development tune-ups is a losing battle, and the problem is more effectively combated further upstream.

Performance Priorities

It is important that some general performance goals are communicated to the development team before development begins. A development task can fall into three broad categories of performance priority.

Performance Insensitive

In this mode, performance really does not matter at all. The software can end up quite slow with no real adverse consequences, and even naive performance mistakes are acceptable. Software that fits into this category is typically single- or special-use, such as a program that will perform a once-off data migration task. If the software ends up slow, the task can be run on a weekend or on a spare machine; in this case development time and quality issues are much more important than speed.

There is nothing inherently wrong with this development mode—in a number of project types, it is much better for a client or manager when the software comes in twice as cheap but ten times as slow. It is important that developers can swallow their pride in this mode, and appreciate the pressures and priorities under which the software is being developed.

Performance Sensitive

This is the normal development mode for most applications. In this mode, producing software that is "not too slow" is the goal, though other development priorities may be more important than performance. Producing detailed metrics that describe how fast "not too slow" is can be hard to achieve, but a few simple goals can be defined to allow a common understanding of the performance targets for the application. For end-user applications, response time and memory usage are two criteria that are easy for all stakeholders to understand, and can be defined as they relate to a target hardware environment. There will usually be application actions that fall outside the response time goals, and the remedies required in terms of both the product (for example, adding a progress bar) and the project (for example, noting the issue in a project manual) should be defined.

Server-side application performance is usually defined in terms of client load, average response time, and, optionally, the response time at some statistical distance from the mean. The expected client load can be difficult for nontechnical stakeholders to appreciate, and there is often a tendency to propose an exaggerated figure. The cost of scalable software design and implementation, particularly on the .NET platform, is not greatly more than a nonscalable design, which leaves hardware, server licenses, and other operational costs as the price delta in supporting an increased load. These costs are easy to quantify, and nontechnical stakeholders generally have no problem in making cost-based decisions.

With some effort, a realistic understanding of hardware requirements and response time can be established with the client, and easily measurable performance goals can be established for the development and testing teams.

Performance Critical/Real Time

This is a rare development mode for Windows applications, and even rarer for .NET applications. The Windows NT family of operating systems can be considered a soft real-time operating system, where a timely response can be expected most of the time, and the average response time is dependable. Venturcom (`http://www.vci.com`) has released a series of real-time extensions for Windows XP that can support hard real-time operation, in which the timeliness of a response is guaranteed 100 percent of the time.

For .NET applications, the common language runtime (CLR) adds a layer of indeterminacy that results in the real-time functionality of the OS being diminished below the soft real-time barrier. During a garbage collection, there is a point at which all managed code is suspended, which has a significant impact on the ability of managed code to guarantee any type of response time. Other factors also affect the runtime's ability to guarantee response times, including Just-In-Time (JIT) compilation and module loading, though these events can have determinant performance characteristics in a closed system. Given these factors, even soft real-time systems are not advisable on the .NET platform. As was seen with Java, it is likely that future Common Language Infrastructure (CLI) implementations will be developed that guarantee some real-time capabilities, and that performance-critical development will be feasible using .NET.

Regardless of platform capabilities, real-time development is a complex and difficult topic, and beyond the scope of this book. For readers interested in real-time software development, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* by Bruce Powel Douglass (Addison-Wesley, 1999) is an excellent reference. Although real-time development is not feasible in the short term, high-performance development is possible with the .NET platform, and CLR and Framework enhancements are likely to deliver performance characteristics that rival native code as .NET matures.

Wrap-Up

By spending a brief amount of time during the design phase of a project in identifying the priority of performance, the specifications of the hardware environment, and some basic performance criteria, the probability of project success is greatly enhanced. Not only do all parties gain a consistent expectation about how the software should perform, but performance risk areas are also more likely to be identified, which allows them to be dealt with early during the design and development phases.

Test Environment and Practices

When developing software where performance is a high priority, performance testing is a critical activity. There are three rules for performance testing:

- Secure access to the target hardware environment early.
- Test early.
- Test often.

For applications where performance is not a high priority, performance testing during a development iteration is not necessary, particularly if the application uses standard architectural patterns and contains no major risk areas. Overeager optimization can lead to complexity and development delays, and should generally be avoided.

For applications where performance is a priority, access to test hardware is an important element of project success. The all-too-common occurrence of the test machine arriving a week after the system is deployed is a recipe for performance headaches. Development machines are usually high-end boxes with plenty of RAM and fast CPUs, and it is impossible to accurately extrapolate performance results of code running on a high-end machine to the results that may be expected on the target hardware environment. If an application is memory intensive, memory paging may destroy the performance when the code is run on the target platform, and this performance problem will not be apparent during unit testing on a high-end development machine. If an exact replication of the target environment is not available, scrounging up a machine of the approximately correct configuration should be possible in most circumstances. If a low-end test machine is not available, removing RAM and disabling performance-oriented hardware features via the BIOS or motherboard jumpers will allow some indication of application performance on low-end machines to be achieved. If hardware modifications are being attempted, ensure that the

appropriate safety practices are followed, and take care not to damage either the hardware or the software engineer—both can be expensive to replace.

In addition to processor and memory similarities, peripheral hardware relevant to the particular application, such as video adapters for video-intensive applications and hard disks for high-IO applications, should resemble the target environment. If the application is to be deployed on legacy operating systems such as Windows 98 or Windows ME, it is critical that the application is also tested on these platforms, as the CLR implementation has many subtle differences that can result in performance issues that do not exist on Windows 2000/XP.

Once a reasonable hardware environment has been established for performance testing, it is important to develop adequate test scenarios, with representative system inputs, and preferably automated test cases. Many automated test tools exist, and a number of excellent free tools are available.[1] It is important to establish the test environment early on in the project, and test the product periodically. The early versions of an application that contain skeleton components with stubbed-out methods will run much quicker than the full release, and if performance is poor in the early releases, architectural problems exist, and it is critical to address these problems early.

Unit testing tools, which are typically used for testing the logical correctness of code, can also be used to test the performance correctness of code. The correctness of a method is generally determined by a Boolean expression at the end of a test case, and timing tests can be used instead of the usual logic tests to ensure that code has the correct performance characteristics.

In addition to preventing unpleasant surprises late in the development cycle, regular performance testing makes identifying performance bugs much easier, and reduces the need for many of the tools covered in the following chapters. A marked decrease in performance between two test runs indicates that code that has been added to the system between tests has introduced or triggered the problem, and this makes tracking down the culprit code much easier than starting from scratch.

Developer Responsibilities

The individual responsibilities of developers in achieving the correct performance characteristics of a software system are often not properly considered. In a naive process-engineering view of software development, developers are often classed as generic resources that need to be added to a task in a cookbook-style approach. This is profoundly incorrect, and the skills and knowledge that developers

1. One of the most popular test tools is NUnit, which is a free unit testing tool available from `http://www.nunit.org`.

possess are reflected in the code they produce. It is the developer's *individual* responsibility to be sufficiently knowledgeable regarding the Framework and software principles to allow for the production of code that has reasonable performance characteristics. The developer does not need to possess a profound knowledge of the internals of the CLR, but should be familiar enough with the product to select the appropriate tools to correctly code a system component. For example, the developer should know that `System.Text.StringBuilder` is better than `System.String` for performing multiple string manipulations, that objects that encapsulate unmanaged resources need explicit cleanup, and that different collection classes have different performance characteristics for insertion, deletion, and searching.

In addition to the numerous books on .NET, the MSDN Library is an excellent resource for learning about the various features of the .NET Framework. The .NET Framework SDK documentation (`http://msdn.microsoft.com/library/en-us/cpguide/html/cpcongettingstartedwithnetframework.asp`) provides detailed coverage of a wide range of topics, and also includes many samples that demonstrate the use of .NET technologies in various languages.

Conclusion

The .NET Framework has a clean and consistent design and implementation that reduces the possibility of producing poorly performing software. Like any new technology, the Framework is not a silver bullet for performance headaches, and having clear development priorities, and supporting these priorities with ongoing testing and monitoring, is an important component in avoiding costly project or product delays caused by slow systems.