

Performance and Security Advantages of Using Managed Code for .NET Applications

This document discusses the benefits and advantages of using managed code in .NET applications. Managed code runs in the Common Language Runtime (CLR), which provides services such as automatic memory management, platform-neutrality, and cross-language integration.

In contrast, unmanaged code does not run inside the .NET environment and cannot use any .NET managed facilities. Unmanaged code includes all code written before the .NET Framework was introduced.

Key features available to managed code applications include:

- Performance benefits gained from executing all code in the CLR. Calling unmanaged code decreases performance because additional security checks are required. Other performance advantages are available through judicious use of the Just-In-Time compiler and NGEN utility.
- Automatic lifetime control of objects, which includes garbage collection and scalability features.
- Ease of deployment and the vastly improved versioning facilities—the end of “DLL hell”.
- Built-in security by using code access security and avoiding buffer overruns.

NOTE: Code examples in this document use the ADO.NET 2.0 Common Programming Model and MetaData capabilities introduced in the Microsoft .NET 2.0 Framework. If you are using an earlier version of the .NET Framework or the DataDirect Connect[®] for .NET 2.2 data providers, refer to [“The Importance of Managed Code in .NET Applications”](#).

Introduction

What is managed code and why is it important to use 100% managed code in .NET applications?

Managed code is compiled for the .NET run-time environment. It runs in the Common Language Runtime (CLR), which is the heart of the .NET Framework. The CLR provides services such as security, memory management, and cross-language integration. (3) Managed applications written to take advantage of the features of the CLR perform more efficiently and safely, and take better advantage of developers' existing expertise in languages that support the .NET Framework.

Unmanaged code includes all application code written before the .NET Framework was introduced—this includes code written to use COM, native Win32, and Visual Basic 6. Because it does not run inside the .NET

environment, unmanaged code cannot make use of any .NET managed facilities. (1)

Advantages of Using Managed Code

Managed code runs entirely “inside the sandbox,” meaning that it cannot make calls outside of the .NET Framework. In this way, managed code gets the maximum benefit from the features of the .NET Framework, and this, in turn, permits applications built with managed code to perform more safely and efficiently.

Performance

The CLR was designed from the start to provide good performance. By using 100% managed code, you can take advantage of the numerous built-in services of the CLR to enhance the performance of your managed application. Because of the runtime services and checks that the CLR performs, applications do not need to include separate versions of these services. (10) And by using 100% managed code, you eliminate the performance costs associated with calling unmanaged code.

Just-In-Time Compiler

The CLR never executes Common Intermediate Language (CIL) directly. Instead, the Just-In-Time (JIT) compiler translates CIL into optimized x86 native instructions. (10) That’s why using managed code lets your software run in different environments safely and efficiently. In addition, using machine language lets you take full advantage of the features of the processor the application is running on. For example, when the JIT encounters an Intel processor, the code produced takes advantage of hyper-threading technology. (5)

Another advantage of the JIT is improved performance. The JIT learns when the code does multiple iterations. The runtime is designed to be able to retune the JIT compiled code as your program runs. (2)

NGEN Utility

NGEN.exe is a .NET utility that post-compiles the application at install time. Post-compiling improves start-up performance for managed code, but maintains all the security and stability advantages of managed assemblies, especially when the application uses Windows Forms. Methods are JITed when they are first used, incurring a larger startup penalty if the application calls many methods during start-up. Because Windows Forms uses many shared libraries in the operating system, post-compiling Windows Forms applications usually improves performance. (13)

Post-compiling also makes sure that the application is optimized for the hardware architecture on the machine on which it is being installed.

Maintaining a 100% Managed Code Environment

Only when your .NET application uses components that are built using 100% managed code do you receive the full benefits of the .NET environment.

For example, when accessing data through ADO.NET, using wire protocol .NET data providers lets you preserve your managed code environment because they do not make calls to native Win32 APIs and Client pieces.

The performance advantages of the managed code environment are lost when you (or the components you are using) call unmanaged code. The CLR makes additional checks on calls to the unmanaged or native code, which impacts performance.

Unmanaged code includes the database client pieces that some .NET data providers require. Examples of .NET data providers that use both managed and unmanaged code are IBM's DB2 data provider and the Oracle Data Provider for .NET (ODP.NET). Both of these data providers must use client libraries to access the database. The data providers shipped by Microsoft for SQL Server and Oracle—as well as the Microsoft OLE DB data providers, and ODBC.NET—make calls to native Win32 database client pieces or other unmanaged code.

Another advantage of using managed code is that you can use the ClickOnce technology of the .NET Framework (see “Ease of Deployment” on page 6). However, *only* 100% managed code solutions can use these deployment methods effectively.

Automatic Memory Management

Automatic memory management is one of the most significant features of managed code. The CLR garbage collector automatically frees allocated objects when there are no longer any outstanding references to them. The developer does not need to explicitly free memory assigned to an object, which goes a long way toward reducing the amount of time spent debugging memory leaks. (11) There can be no memory leaks in 100% managed code.

In addition, using the NGEN.exe utility to post-compile the application at install time saves memory because the JIT compiler is not running, and also preserves shared memory.

Automatic Lifetime Control of Objects

Another significant advantage of using managed code is that the CLR provides automatic lifetime management of components and modules. Lifetime control includes:

- Garbage collection, which frees and releases memory.
- Scalability features, such as thread pooling and the ability to use a non-persistent connection with a dataset.
- Support for side-by-side versions.

Garbage Collection

When an object is created with the `new` operator, the runtime allocates memory from the managed heap. Periodically, the CLR garbage collector checks the heap and automatically disposes of any objects that are no longer being used by the application, reclaiming their memory.

The garbage collector also compacts the released memory, reducing fragmentation. (4) This function is particularly important when the application runs on large memory servers. Changing the application to use smaller objects can help to improve the effectiveness of the garbage collector. Similarly, because each DLL is assigned a 64-bit chunk of memory, combining small DLLs avoids inefficient use of memory.

Because the garbage collector automatically closes unused objects, memory leaks are not possible in an application that uses 100% managed code.

Good coding practices dictate that `Close` and `Dispose` should be used when finished with resources so that the garbage collector can function.

Scalability Features

Thread pooling lets you make much more efficient use of multiple threads and is an important scalability feature of using managed code. The .NET Framework comes with built-in support for creating threads and using the system-provided thread pool. In particular, the `ThreadPool` class under the `System.Threading` namespace provides static methods for submitting requests to the thread pool. In managed code, if one of the threads becomes idle, the thread pool injects another worker thread into the multithread apartment to keep all the processors busy.

The standard `ThreadPool` methods capture the caller's stack and merge it into the stack of the thread-pool thread when the thread-pool thread starts to execute a task. If you are using unmanaged code, the entire stack will be checked, which incurs a performance cost. In some cases, you can eliminate the stack checking with the `Unsafe` methods `ThreadPool.UnsafeQueueUserWorkItem` and `ThreadPool.UnsafeRegisterWaitForSingleObject`, which provide better performance. However, using the `Unsafe` method calls does not provide complete safety. (8)

Further adding to scalability is the ability to use a non-persistent connection with a dataset, which is a cache of the records retrieved from the database. The dataset keeps track of the state of the data and stores the data as pure XML. Database connections are opened and closed only as needed to retrieve data into the dataset, or to return updated data. (7)

Versioning

Versioning essentially eliminates "DLL hell." When you define an assembly as strongly named, the .NET executable will be executed with the same DLL with which it was built. This means that you can have side-by-side versions

of a DLL, allowing you to manage shared components. Versioning ensures that each time an application starts up, it checks its shared files. If a file has changed and the changes are incompatible, the application can ask the runtime for a compatible version.

However, when an application calls unmanaged DLLs, you can end up back in "DLL hell." For example, Oracle's ODP.NET data provider calls the unmanaged Oracle Client pieces, which are specific to a particular version of Oracle. You could install two versions of this *unmanaged* data provider, for example, one for Oracle9i and one for the upcoming Oracle10g, but you would have a conflict, because each data provider will require a particular version of the clients. Since the clients are native Win32 DLLs, you cannot easily have side-by-side versions running on the same machine. Only with native wire protocol data providers built from 100% managed code can you install side-by-side versions with no configuration required by the end-user.

Checks by the .NET Runtime

The .NET runtime automatically performs numerous checks to ensure that code is written correctly. Because these checks prevent a large number of bugs from ever happening, developer productivity is improved and the application quality is better. In addition, these checks thwart system attacks such as the exploitation of buffer overruns.

The CLR checks for type safety to ensure that applications always access allocated objects in appropriate ways. In other words, if a method input parameter is declared as accepting a 4-byte value, the common language runtime will detect and trap attempts to access the parameter as an 8-byte value. Type safety also means that execution flow will only transfer to known method entry points. There is no way to construct an arbitrary reference to a memory location and cause code at that location to begin execution.

In addition, array indexes are checked to be sure they are in the range of the array. For example, if an object occupies 10 bytes in memory, the application can't change the object so that it will allow more than 10 bytes to be read.

(12)

Cross-language Integration

You can write .NET applications in many different languages, such as C, C++, Visual Basic, COBOL, Fortran, Perl, Pascal, Jscript, Lisp, Python, Smalltalk, and others. Programmers can use the languages in which they are most proficient to develop portions of an application.

All CLR-compliant languages compile to Common Intermediate Language (CIL). CIL is the key to making the .NET application platform-neutral and hardware independent.

In addition, programmers can choose specific languages for specific tasks within the same application. Some languages are stronger than others for particular tasks, and programmers can choose the language best suited for

