Startup Performance

For client applications, quick startup is a great first impression to make on the user or a potential client evaluating the product or taking a demo. However, the more complex the application, the harder it is to maintain acceptable startup times. It's important to differentiate between *cold startup*, when the application is launched for the first time after the system has just finished booting, and *warm startup*, when the application is launched (not for the first time) after the system has been used for some time. Always-on system services or background agents need fast cold startup times, so as to prevent the system startup sequence and user logon from taking too long. Typical client applications such as email clients and Web browsers might get away with slightly longer cold startup times, but users will expect them to launch rapidly when the system has been used for a while. Most users will want short startup times in both cases.

There are several factors that go into long startup times. Some of them are relevant only for cold startup; others are relevant for both types of startup.

- I/O operations—to launch the application, Windows and the CLR must load from the disk the application's assemblies as well as .NET Framework assemblies, CLR DLLs, and Windows DLLs. This factor is relevant mostly for cold startup.
- JIT compilation—every method called for the first time during application startup must be compiled by the JIT compiler. Because code compiled by the JIT is not preserved when the application terminates, this factor is relevant for both cold and warm startup.
- GUI initialization—depending on your GUI framework (Metro, WPF, Windows Forms, etc.), there are GUI-specific initialization steps that must take place for the user interface to be displayed. This factor is relevant for both types of startup.
- Loading application-specific data—your application might require some information from files, databases, or Web services to display its initial screen. This factor is relevant for both types of startup, unless your application employs some caching scheme for this data.

We have several measurement tools at our disposal to diagnose long startup times and their likely causes (see Chapter 2). Sysinternals Process Monitor can point to I/O operations performed by the application's process, regardless of whether Windows, the CLR, or the application code initiated them. PerfMonitor and the `.NET CLR JIT` performance counter category can help diagnose excessive JIT compilation during application startup. Finally, "standard" profilers (in sampling or instrumentation modes) can determine how your application is spending its time during startup.

Additionally, there is a simple experiment you can perform that will tell you whether I/O is the primary cause of slow startup times: time your application's cold startup and warm startup scenarios (you should use a clean hardware environment for these tests, and make sure there are no unnecessary services or other initialization taking place in the cold startup test). If the warm startup is significantly faster than the cold startup, I/O is the chief culprit.

Improving your application-specific data loading is up to you; every guideline we can issue would be too general to use in practice (other than providing a splash screen to stretch your users' patience…). However, we can offer several remedies for poor startup performance that originates from I/O operations and JIT compilation. Optimizing your application's startup time can cut it down in half or more, in some cases.

Pre-JIT Compilation with NGen (Native Image Generator)

Although the JIT compiler is very convenient and compiles methods only when they are called, there is still a performance cost that your application pays whenever the JIT runs. The .NET Framework offers an optimization tool, called *Native Image Generator* (NGen.exe), which can compile your assemblies to machine code (*native images*) before runtime. If every assembly your application requires is pre-compiled in this manner, there will be no need to load the JIT compiler and to use it during application startup. Although the generated native image is likely to be larger than the original assembly, in most cases the amount of disk I/O on cold startup actually decreases, because the JIT compiler itself (clrjit.dll) and metadata for referenced assemblies aren't read from disk.

Pre-compilation has another advantageous effect—native images can be shared between processes, unlike code that the JIT compiler emits at runtime. If several processes on the same machine use a native image for some assembly, physical memory consumption is lower than in the JIT-compiled case. This is especially important in shared-system scenarios, when multiple users connect through Terminal Services sessions to a single server, and run the same application.

To pre-compile your application, all you need to do is point the NGen.exe tool, which resides in the .NET Framework's directory, to your application's main assembly (typically an .exe file). NGen will proceed to locate all the static dependencies your main assembly has and pre-compile them all to native images. The resulting native images are stored in a cache that you do not have to manage—it is stored next to the GAC, in the `C:\Windows\Assembly\NativeImages_*` folders by default.

Tip Because the CLR and NGen manage the native image cache automatically, you should *never* copy the native images from one machine to another. The only supported way to pre-compile managed assemblies on a specific system is to run the NGen tool on *that* system. The ideal time to do this would be during the application's installation (NGen even supports a "defer" command, which will queue pre-compilation to a background service). This is what the. NET Framework installer does for frequently used .NET assemblies.

Below is a complete example of using NGen to pre-compile a simple application that consists of two assemblies—the main .exe file, and an auxiliary .dll referenced by it. NGen successfully detects the dependency and pre-compiles both assemblies to native images:

```
> c:\windows\microsoft.net\framework\v4.0.30319\ngen install Ch10.exe
Microsoft (R) CLR Native Image Generator - Version 4.0.30319.17379
Copyright (c) Microsoft Corporation. All rights reserved.

Installing assembly D:\Code\Ch10.exe
1> Compiling assembly D:\Code\Ch10.exe (CLR v4.0.30319) . . .
2> Compiling assembly HelperLibrary, . . . (CLR v4.0.30319) . . .
```

At runtime, the CLR uses the native images and does not require clrjit.dll to be loaded at all (in the `lm` command's output below, clrjit.dll is not listed). Type method tables (see Chapter 3), also stored in the native images, point to the pre-compiled versions inside the native image boundaries.

```
0:007 > lm

start end module name
01350000 01358000 Ch10 (deferred)
```

```
2f460000 2f466000 Ch10_ni (deferred)
30b10000 30b16000 HelperLibrary_ni (deferred)
67fa0000 68eef000 mscorlib_ni (deferred)
6b240000 6b8bf000 clr (deferred)
6f250000 6f322000 MSVCR110_CLR0400 (deferred)
72190000 7220a000 mscoreei (deferred)
72210000 7225a000 MSCOREE (deferred)
74cb0000 74cbc000 CRYPTBASE (deferred)
74cc0000 74d20000 SspiCli (deferred)
74d20000 74d39000 sechost (deferred)
74d40000 74d86000 KERNELBASE (deferred)
74e50000 74f50000 USER32 (deferred)
74fb0000 7507c000 MSCTF (deferred)
75080000 7512c000 msvcrt (deferred)
75150000 751ed000 USP10 (deferred)
753e0000 75480000 ADVAPI32 (deferred)
75480000 75570000 RPCRT4 (deferred)
75570000 756cc000 ole32 (deferred)
75730000 75787000 SHLWAPI (deferred)
75790000 757f0000 IMM32 (deferred)
76800000 7680a000 LPK (deferred)
76810000 76920000 KERNEL32 (deferred)
76920000 769b0000 GDI32 (deferred)
775e0000 77760000 ntdll (pdb symbols)

0:007 > !dumpmt -md 2f4642dc
EEClass: 2f4614c8
Module: 2f461000
Name: Ch10.Program
mdToken: 02000002
File: D:\Code\Ch10.exe
BaseSize: 0xc
ComponentSize: 0x0
Slots in VTable: 6
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
   Entry MethodDe JIT Name
68275450 68013524 PreJIT System.Object.ToString()
682606b0 6801352c PreJIT System.Object.Equals(System.Object)
68260270 6801354c PreJIT System.Object.GetHashCode()
68260230 68013560 PreJIT System.Object.Finalize()
2f464268 2f46151c PreJIT Ch10.Program..ctor()
2f462048 2f461508 PreJIT Ch10.Program.Main(System.String[])

0:007 > !dumpmt -md 30b141c0
EEClass: 30b114c4
Module: 30b11000
Name: HelperLibrary.UtilityClass
mdToken: 02000002
File: D:\Code\HelperLibrary.dll
BaseSize: 0xc
ComponentSize: 0x0
Slots in VTable: 6
Number of IFaces in IFaceMap: 0
--------------------------------------
MethodDesc Table
   Entry MethodDe JIT Name
68275450 68013524 PreJIT System.Object.ToString()
682606b0 6801352c PreJIT System.Object.Equals(System.Object)
68260270 6801354c PreJIT System.Object.GetHashCode()
```

```
68260230 68013560 PreJIT System.Object.Finalize()
30b14158 30b11518 PreJIT HelperLibrary.UtilityClass..ctor()
30b12048 30b11504 PreJIT HelperLibrary.UtilityClass.SayHello()
```

Another useful option is the "update" command, which forces NGen to recalculate the dependencies for all native images in the cache and pre-compile again any modified assemblies. You would use this after installing an update on the target system, or during development.

Note Theoretically, NGen could use a completely different—and larger—set of optimizations than what the JIT compiler uses at runtime. After all, NGen is not as time-bound as the JIT compiler. However, at the time of writing, NGen has no optimizations up its sleeve other than those the JIT compiler employs.

When using CLR 4.5 on Windows 8, NGen does not passively wait for your instructions to pre-compile application assemblies. Instead, the CLR generates assembly usage logs that are processed by NGen's background maintenance task. This task decides, periodically, which assemblies would benefit from pre-compilation, and runs NGen to create native images for them. You can still use NGen's "display" command to review the native image cache (or inspect its files from a command line prompt), but much of the burden of deciding which assemblies would benefit from pre-compilation is now at the CLR's discretion.

Multi-Core Background JIT Compilation

Starting in CLR 4.5, you can instruct the JIT compiler to generate a profile of methods executed during application startup, and use that profile on subsequent application launches (including cold startup scenarios) to compile these methods in the background. In other words, while your application's main thread is initializing, the JIT compiler executes on background threads such that methods are likely to be already compiled by the time they are needed. The profile information is updated on each run, so it will remain fresh and up to date even if the application is launched for hundreds of times with different configurations.

Note This feature is enabled by default in ASP.NET and Silverlight 5 applications.

To opt into using multi-core background JIT compilation, you call two methods on the `System.Runtime.ProfileOptimization` class. The first method tells the profiler where the profiling information can be stored, and the second method tells the profiler which startup scenario is being executed. The purpose of the second method is to distinguish between considerably different scenarios, such that the optimization is tailored for the specific scenario. For example, an archiving utility may be invoked with a "display files in archive" parameter, which requires one set of methods, and a "create compressed archive from directory" parameter, which requires a completely different set of methods:

```
public static void Main(string[] args) {
  System.Runtime.ProfileOptimization.SetProfileRoot(
    Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location));
  if (args[0] == "display") {
    System.Runtime.ProfileOptimization.StartProfile("DisplayArchive.prof");
  } else if (args[0] == "compress") {
    System.Runtime.ProfileOptimization.StartProfile("CompressDirectory.prof");
  }
  //. . .More startup scenarios
  //The rest of the application's code goes here, after determining the startup scenario
}
```

Image Packers

A common approach to minimizing I/O transfer costs is by compressing the original data. After all, it would make little sense to download a 15GB uncompressed Windows installation if it can fit—in compressed form—on a single DVD. The same idea can be used with managed applications stored on disk. Compressing the application and decompressing it only after loading it into memory can significantly decrease cold startup costs by reducing the number of I/O operations performed on startup. Compression is a double-edged sword, because of CPU time required to decompress the application's code and data in memory, but it can be worthwhile to pay the price in CPU time when reducing cold startup costs is critical, such as with a kiosk application that has to launch as quickly as possible after the system is turned on.

There are several commercial and open source compression utilities for applications (usually called *packers*). If you use a packer, make sure it can compress .NET applications—some packers will only work well with unmanaged binaries. One example of a tool that compresses .NET applications is MPress, which is freely available online at http://www.matcode.com/mpress.htm. Another example is the Rugland Packer for .NET Executables (RPX), an open source utility published at http://rpx.codeplex.com/. Below is some sample output from RPX when run on a very small application:

```
> Rpx.exe Shlook.TestServer.exe Shlook.Common.dll Shlook.Server.dll
Rugland Packer for (.Net) eXecutables 1.3.4399.43191

                                                                    100.0%

Unpacked size :.............27.00 KB
 Packed size :.............13.89 KB
 Compression :.............48.55%
 ---------------------------------------

 Application target is the console
 ---------------------------------------

 Uncompressed size :.............27.00 KB
 Startup overhead :..............5.11 KB
 Final size :.............19.00 KB
 ---------------------------------------

 Total compression :.............29.63%
```

Managed Profile-Guided Optimization (MPGO)

Managed profile guided optimization (MPGO) is a tool introduced in Visual Studio 11 and CLR 4.5 that optimizes the on-disk layout of native images produced by NGen. MPGO generates profile information from a specified period of the application's execution and embeds it in the assembly. Subsequently, NGen uses this information to optimize the layout of the generated native image.

MPGO optimizes native image's layout in two primary ways. First, MPGO makes sure that code and data which are used frequently (hot data) are placed together on disk. As a result, there will be fewer page faults and disk reads for hot data, because more hot data fits on a single page. Second, MPGO places potentially writable data together on disk. When a data page that is shared with other processes is modified, Windows creates a private copy of the page for the modifying process to use (this is known as *copy-on-write*). As a result of MPGO's optimization, fewer shared pages are modified, resulting in less copies and lower memory utilization.

To run MPGO on your application, you need to provide the list of assemblies to instrument, an output directory in which to place the optimized binaries, and a timeout after which profiling should stop. MPGO instruments the application, runs it, analyzes the results, and creates optimized native images for the assemblies you specified:

```
> mpgo.exe -scenario Ch10.exe -assemblylist Ch10.exe HelperLibrary.dll -OutDir . -NoClear

Successfully instrumented assembly D:\Code\Ch10.exe
Successfully instrumented assembly D:\Code\HelperLibrary.dll

 < output from the application removed >

Successfully removed instrumented assembly D:\Code\Ch10.exe
Successfully removed instrumented assembly D:\Code\HelperLibrary.dll
Reading IBC data file: D:\Code\Ch10.ibc
The module D:\Code\Ch10-1.exe, did not contain an IBC resource
Writing profile data in module D:\Code\Ch10-1.exe
Data from one or more input files has been upgraded to a newer version.
Successfully merged profile data into new file D:\Code\Ch10-1.exe
Reading IBC data file: D:\Code\HelperLibrary.ibc
The module D:\Code\HelperLibrary-1.dll, did not contain an IBC resource
Writing profile data in module D:\Code\HelperLibrary-1.dll
Data from one or more input files has been upgraded to a newer version.
Successfully merged profile data into new file D:\Code\HelperLibrary-1.dll
```

Note When the optimization completes, you need to run NGen again on the optimized assemblies to create final native images that benefit from MPGO. Running NGen on assemblies is covered earlier in this chapter.

At the time of writing, there were no plans to introduce MPGO into the Visual Studio 2012 user interface. The command line tool is the only way to add these performance gains to your application. Because it relies on NGen, this is another optimization that is best performed after installation time on the target machine.

Miscellaneous Tips for Startup Performance

There are several additional tips we didn't mention earlier that might be able to shave a few more seconds off your application's startup time.

Strong Named Assemblies Belong in the GAC

If your assemblies have strong names, make sure to place them in the global assembly cache (GAC). Otherwise, loading the assembly requires touching almost every page to verify its digital signature. Verification of strong names when the assembly is not placed in the GAC will also diminish any performance gains from NGen.

Make Sure Your Native Images Do Not Require Rebasing

When using NGen, make sure that your native images do not have base address collisions, which require rebasing. Rebasing is an expensive operation that involves modifying code addresses at runtime, and creates copies of otherwise shared code pages. To view the native image's base address, use the dumpbin.exe utility with the /headers flag, as follows:

```
> dumpbin.exe /headers Ch10.ni.exe
Microsoft (R) COFF/PE Dumper Version 11.00.50214.1
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file Ch10.ni.exe

PE signature found

File Type: DLL

FILE HEADER VALUES
          14C machine (x86)
          4 number of sections
```

```
        4F842B2C time date stamp Tue Apr 10 15:44:28 2012
            0 file pointer to symbol table
            0 number of symbols
            E0 size of optional header
            2102 characteristics
            Executable
            32 bit word machine
            DLL

  OPTIONAL HEADER VALUES
            10B magic # (PE32)
            11.00 linker version
            0 size of code
            0 size of initialized data
            0 size of uninitialized data
            0 entry point
            0 base of code
            0 base of data
        30000000 image base (30000000 to 30005FFF)

            1000 section alignment
            200 file alignment
            5.00 operating system version
            0.00 image version
            5.00 subsystem version
            0 Win32 version
            6000 size of image
   < more output omitted for brevity>
```

To change the native image's base address, change the base address in the project properties in Visual Studio. The base address can be located on the Build tab, after opening the Advanced dialog (see Figure 10-1).

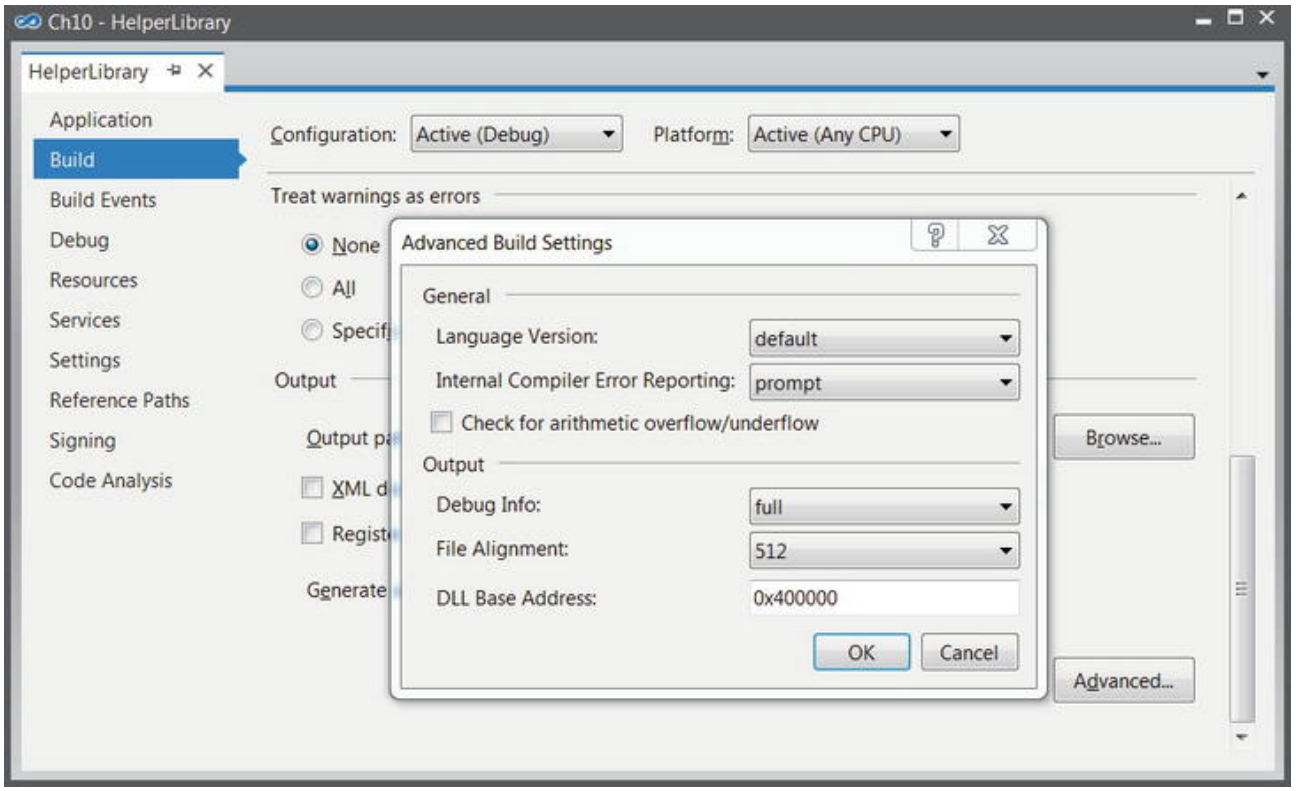

Figure 10-1. Visual Studio Advanced Build Settings dialog, which allows modifying the base address for the native image that NGen generates

As of .NET 3.5 SP1, NGen opts-in to use Address Space Layout Randomization (ASLR) when the application is running on Windows Vista or a newer platform. When using ASLR, base load addresses for images are randomized across runs for security reasons. Under this configuration, rebasing assemblies to avoid base address collisions is not important on Windows Vista and newer platforms.

Reduce the Total Number of Assemblies

Reduce the number of assemblies loaded by your application. Each assembly load incurs a constant cost, regardless of its size, and inter-assembly references and method calls may be more costly at runtime. Large applications that load hundreds of assemblies are not uncommon, and their startup times can be reduced by several seconds if assemblies are merged to only a few binaries.