Writing High-Performance Managed Applications : A Primer

15 out of 20 rated this helpful

Gregor Noriskin
Microsoft CLR Performance Team

June 2003

Applies to:
  Microsoft® .NET Framework

Summary: Learn about the .NET Framework's Common Language Runtime from a performance perspective. Learn how to identify managed code performance best practices and how to measure the performance of your managed application. (19 printed pages)

Download the CLR Profiler. (330KB)

Contents

Juggling as a Metaphor for Software Development

Juggling is a great metaphor for describing the software development process. Juggling typically requires at least three items, though there is no upper limit to the number of items you can attempt to juggle. When you begin learning how to juggle you find that you watch each ball individually as you catch and throw them. As you progress you begin to focus on the flow of the balls, as opposed to each individual ball. When you have mastered juggling, you can once again concentrate on a single ball, balancing that ball on your nose, while continuing to juggle the others. You know intuitively where the balls are going to be and can put your hand in the right place to catch and throw them. So how is this like software development?

Different roles in the software development process juggle different "trinities"; Project and Program Managers juggle Features, Resources and Time, and Software Developers juggle Correctness, Performance and Security. One can always try to juggle more items, but as any student of juggling can attest to, adding a single ball makes it exponentially more difficult to keep the balls in the air. Technically if you are juggling less than three balls you are not juggling at all. If, as a software developer, you are not considering the correctness, performance and security of the code you are writing, the case can be made that you are not doing your job. When you initially start considering Correctness, Performance and Security, you will find yourself having to focus on one aspect at a time. As they become part of your day-to-day practice you will find that you do not need to focus on a specific aspect, they will simply be part of the way you work. When you have mastered them you will be able to intuitively make tradeoffs, and focus your efforts appropriately. And as with juggling, practice is the key.

Writing high-performance code has a trinity of its own; Setting Goals, Measurement and Understanding the Target Platform. If you do not know how fast your code has to be, how will you know when you are done? If you do not measure and profile your code, how will you know when you have met your goals, or why you are not meeting your goals? If you do not understand the platform you are targeting, how will you know what to optimize in the case that you are not meeting your goals. These principles apply to the development of high-performance code in general, whichever platform you are targeting. No article on writing high-performance code would be complete without mentioning this trinity. Though all three are equally significant, this article is going to focus on the latter two aspects as they apply to writing high-performance applications that target the Microsoft® .NET Framework.

The fundamental principles of writing high-performance code on any platform are:

1. Set Performance Goals
2. Measure, measure, and then measure some more
3. Understand the hardware and software platforms your application is targeting

The .NET Common Language Runtime

The core of the .NET Framework is the Common Language Runtime (CLR). The CLR provides all the runtime services for your code; Just-In-Time compilation, Memory Management, Security and a number of other services. The CLR was designed to be high-performance. That said there are ways you can take advantage of that performance and ways you can hinder it.

The goal of this article is to give an overview of the Common Language Runtime from a performance perspective, identify managed code performance best practices, and to show how you can measure the performance of your managed application. This article is not an exhaustive discussion on the performance characteristics of the .NET Framework. For the purposes of this article I will define performance to include throughput, scalability, startup time and memory usage.

Managed Data and the Garbage Collector

One of developers' primary concerns about using managed code in performance-critical applications is the cost of the CLR's memory management, which is carried out by the Garbage Collector (GC). The cost of memory management is a function of the allocation cost of memory associated with an instance of a type, the cost of managing that memory over the lifetime of the instance, and the cost of freeing that memory when it is no longer needed.

A managed allocation is usually very cheap; in most cases taking less time than a C/C++ malloc or new. This is because the CLR does not need to scan a free list to find the next available contiguous block of memory big enough to hold the new object; it keeps a pointer to the next free position in memory. One can think of managed heap allocations as being "stack like". An allocation might cause a collection if the GC needs to free memory to allocate the new object, in which case the allocation is more expensive than a malloc or new. Objects that are pinned can also affect the allocation cost. Pinned objects are objects that the GC has been instructed not to move during a collection, typically because the address of the object has been passed to a native API.

Unlike a malloc or new, there is a cost associated with managing memory over the lifetime of an object. The CLR GC is generational, which means that the entire heap is not always collected. However, the GC still needs to know if any live objects in the rest of the heap root objects in the portion of the heap that is being collected. Memory that contains objects that hold references to objects in younger generations is expensive to manage over the lifetime of the objects.

The GC is a Generational Mark and Sweep Garbage Collector. The managed heap contains three generations; Generation 0 contains all new objects, Generation 1 contains slightly longer-lived objects, and Generation 2 contains long-lived objects. The GC will collect the smallest section of the heap possible to free enough memory for the application to continue. The collection of a Generation includes the collection of all younger generations, in this case a Generation 1 collection also collects Generation 0. Generation 0 is dynamically sized according to the size of the processor's cache and the application's allocation rate, and typically takes less than 10 milliseconds to collect. Generation 1 is sized dynamically according to the allocation rate of the application and typically takes between 10 and 30 milliseconds to collect. Generation 2 size will depend on the allocation profile of your application, as will the time it takes to collect. It is these Generation 2 collections that will most significantly affect the performance cost of managing your applications' memory.

> **HINT** The GC is self-tuning and will adjust itself according to applications memory requirements. In most cases programmatically invoking a GC will hinder that tuning. "Helping" the GC by calling **GC.Collect** will more than likely not improve your applications performance.

The GC may relocate live objects during a collection. If those objects are large the cost of relocation is high so those objects are allocated in a special area of the heap called the Large Object Heap. The Large Object Heap is collected, but is not compacted, for example, large objects are not relocated. Large objects are those objects that are larger than 80kb. Note that this may change in future versions of the CLR.

When the Large Object Heap needs to be collected it forces a full collection, and the Large Object Heap is collected during Gen 2 collections. The allocation and death rate of objects in the Large Object Heap can have a significant effect on the performance cost of managing your applications memory.

Allocation Profiles

The overall allocation profile of a managed application will define how hard the Garbage Collector has to work to manage the memory associated with the application. The harder the GC has to work managing the memory, the greater the number of CPU cycles the GC takes, and the less time the CPU is going to spend running the application code. The allocation profile is a function of the number of objects allocated, the size of those objects, and their lifetimes. The most obvious way to relieve GC pressure is simply to allocate fewer objects. Applications designed for extensibility, modularity and reuse using Object Orientated design techniques will almost always result is an increased number of allocations. There is a performance penalty for abstraction and "elegance".

A GC-friendly allocation profile will have some objects allocated at the beginning of the application and then surviving for the lifetime of the application, and then all other objects being short lived. Long-lived objects will contain few or no references to short lived objects. As the allocation profile deviates from this, the GC will have to work harder to manage the applications memory.

A GC-unfriendly allocation profile will have many objects surviving into Generation 2 and then dying, or will have many short-lived objects being allocated in the Large Object Heap. Objects that survive long enough to get into Generation 2 and then die are the most expensive to manage. As I mentioned before objects in older generations that contain references to objects in younger generations during a GC, also increase the cost of the collection.

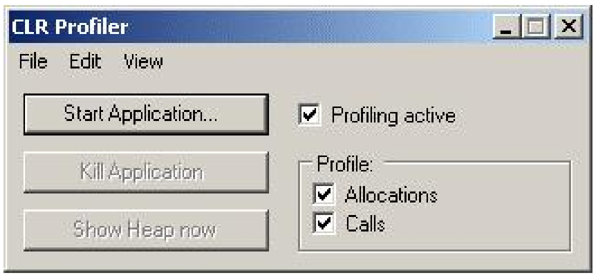
A typical real-world allocation profile will be somewhere in between the two allocation profiles mentioned above. An important metric of your allocation profile is the percentage of the total CPU time that is being spent in GC. You can get this number from the *.NET CLR Memory: % Time in GC* performance counter. If this counter's mean value is above 30% you probably should consider taking a closer look at your allocation profile. This does not necessarily mean that your allocation profile is "bad"; there are some memory intensive applications where this level of GC is necessary and appropriate. This counter should be the first thing you look at if you run into performance problems; it should immediately show if your allocation profile is part of the problem.

HINT If the .NET CLR Memory: % Time in GC performance counter indicates that your application is spending an average of more than 30% of its time in GC you should take a closer look at your allocation profile.

HINT A GC-friendly application will have significantly more Generation 0 than Generation 2 collections. This ratio can be established by comparing the NET CLR Memory: # Gen 0 Collections and NET CLR Memory: # Gen 2 Collections performance counters.

The Profiling API and the CLR Profiler

The CLR includes a powerful profiling API which allows 3rd parties to write custom profilers for managed applications. The CLR Profiler is an unsupported allocation profiling sample tool, written by the CLR Product Team, that uses this profiling API. The CLR Profiler allows developers to see the allocation profile of their manage applications.



☑ Profiling active
Profile:
☑ Allocations
☑ Calls

Figure 1 CLR Profiler Main Window

The CLR Profiler includes a number of very useful views of the allocation profile, including a histogram of allocated types, allocation and call graphs, a time line showing GCs of various generations and the resulting state of the managed heap after those collections, and a call tree showing per-method allocations and assembly loads.

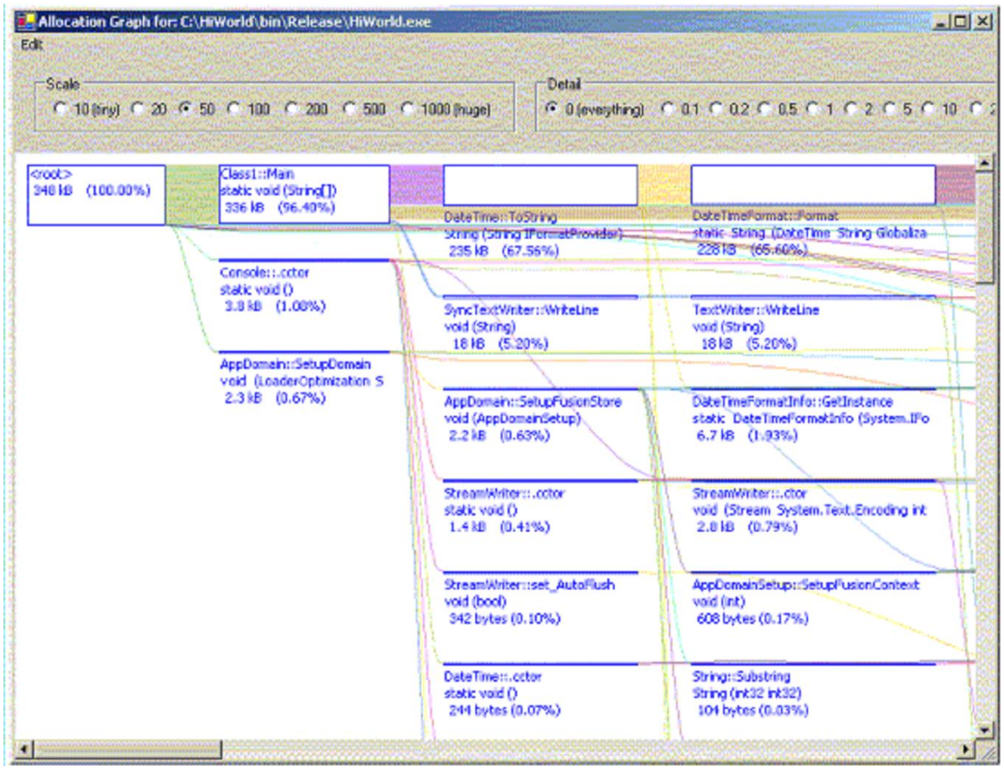

Allocation Graph for: C:\HiWorld\bin\Release\HiWorld.exe
Edit
Scale
10 (tiny) 20 50 100 200 500 1000 (huge)
Detail
0 (everything) 0.1 0.2 0.5 1 2 5 10
<root> 348 kB (100.00%)
Class1::Main static void (String[]) 336 kB (96.40%)
DateTime::ToString String (String IFormatProvider) 235 kB (67.56%)
DateTimeFormat::Format static String (DateTime String Globaliza 228 kB (65.60%)
Console::.cctor static void () 3.8 kB (1.08%)
SyncTextWriter::WriteLine void (String) 18 kB (5.20%)
TextWriter::WriteLine void (String) 18 kB (5.20%)
AppDomain::SetupDomain void (LoaderOptimization S 2.3 kB (0.67%)
AppDomain::SetupFusionStore void (AppDomainSetup) 2.2 kB (0.63%)
DateTimeFormatInfo::GetInstance static DateTimeFormatInfo (System.IFo 6.7 kB (1.93%)
StreamWriter::.cctor static void () 1.4 kB (0.41%)
StreamWriter::.ctor void (Stream System.Text.Encoding int 2.8 kB (0.79%)
StreamWriter::set_AutoFlush void (bool) 342 bytes (0.10%)
AppDomainSetup::SetupFusionContext void (int) 608 bytes (0.17%)
DateTime::.cctor static void () 244 bytes (0.07%)
String::Substring String (int32 int32) 104 bytes (0.03%)

Figure 2 CLR Profiler Allocation Graph

> **HINT** For details on how to use the CLR Profiler see the readme file that is included in the zip.

Note that the CLR Profiler has a high-performance overhead and significantly changes the performance characteristics of your application. Emergent stress bugs will probably disappear when you run your application with the CLR Profiler.

Hosting the Server GC

Two different Garbage Collectors are available for the CLR: a Workstation GC and a Server GC. Console and Windows Forms applications host the Workstation GC, and ASP.NET hosts the Server GC. The Server GC is optimized for throughput and multi-processor scalability. The server GC pauses all threads running managed code for the entire duration of a collection, including both the Mark and Sweep Phases, and GC happens in parallel on all CPU's available to the process on dedicated high-priority CPU-affinitized threads. If threads are running native code during a GC then those threads are paused only when the native call returns. If you are building a server application that is going to run on multiprocessor machines then it is highly recommended that you use the Server GC. If your application in not hosted by ASP.NET, then you are going to have to write a native application that explicitly hosts the CLR.

> **HINT** If you are building scalable server applications, host the Server GC. See Implement a Custom Common Language Runtime Host for Your Managed App.

The Workstation GC is optimized for low latency, which is typically required for client applications. One does not want a noticeable pause in a client application during a GC, since typically client performance is not measured by raw throughput, but rather by perceived performance. The Workstation GC does concurrent GC, which means that it does the Mark Phase while managed code is still running. The GC will only pause threads that are running managed code when it needs to do the Sweep Phase. In the Workstation GC, GC is done on one thread only and therefore on one CPU only.

Finalization

The CLR provides a mechanism whereby clean-up is done automatically before the memory associated with an instance of a type is freed. This mechanism is called Finalization. Typically Finalization is used to release native resources, in this case database connections or operating system handles that are being used by an object.

Finalization is an expensive feature and increases the pressure that is put on the GC. The GC tracks the objects that require finalization in a Finalizable Queue. If during a collection the GC finds an object that is no longer live, but requires finalization, then that object's entry in the Finalizable Queue is moved to the FReachable Queue. Finalization happens on a separate thread called the Finalizer Thread. Because the object's entire state may be required during the execution of the Finalizer, the object and all objects that it points to are promoted to the next generation. The memory associated with the object, or graph of objects, is only freed during the following GC.

Resources that need to be released should be wrapped in as small a Finalizable object as possible; for instance, if your class requires references to both managed and unmanaged resources, then you should wrap the unmanaged resources in a new Finalizable class and make that class a member of your class. The parent class should not be Finalizable. This means that only the class that contains the unmanaged resources will be promoted (assuming you do not hold a reference to the parent class in the class containing the unmanaged resources). Another thing to keep in mind is that there is only one Finalization Thread. If a Finalizer causes this thread to block, the subsequent Finalizers will not be called, resources will not be freed, and your application will leak.

> **HINT** Finalizers should be kept as simple as possible and should never block.
>
> **HINT** Make finalizable only the wrapper class around unmanaged objects that need cleanup.

Finalization can be thought of as an alternative to reference counting. An object that implements reference counting keeps track of how many other objects have references to it (which can lead to some very well known problems), so that it can release its resources when its reference count is zero. The CLR does not implement reference counting so it needs to provide a mechanism to automatically release resources when no more references to the object are being held. Finalization is that mechanism. Finalization is typically only needed in the case where the lifetime of an object that requires clean-up is not explicitly known.

The Dispose Pattern

In the case that the lifetime of the object is explicitly known, the unmanaged resources associated with an object should be eagerly released. This is called "Disposing" the object. The Dispose Pattern is implemented through the IDisposable interface (though implementing it yourself would be trivial). If you wish to make eager finalization available for your class, for example, make instances of your class disposable, you need to have your object implement the IDisposable interface and provide an implementation for the **Dispose** method. In the **Dispose** method you will call the same cleanup code that is in the Finalizer and inform the GC that it no longer needs to finalize the object by calling the **GC.SuppressFinalization** method. It is a good practice to have both the **Dispose** method and the Finalizer call a common finalization function so that only one version of the clean-up code needs to be maintained. Also, if the semantics of the object are such that a **Close** method will be more logical than a **Dispose** method then a **Close** should also be implemented; in this case a database connection or a socket are logically "closed". The **Close** can simply call the **Dispose** method.

It is always a good practice to provide a Dispose method for classes with a Finalizer; one can never be sure how that class is going to be used, for instance, whether its lifetime is going to be explicitly known or not. If a class that you are using implements the Dispose pattern and you explicitly know when you are done with the object, most definitely call **Dispose**.

> **HINT** Provide a **Dispose** method for all classes that are finalizable.
>
> **HINT** Suppress Finalization in your **Dispose** method.
>
> **HINT** Call a common cleanup function.
>
> **HINT** If an object that you are using implements IDisposable and you know that the object is no longer needed, call Dispose.

C# provides a very convenient way to automatically dispose objects. The using keyword allows you to identify a block of code after which **Dispose** will be called on a number of disposable objects.

C#'s using keyword

```
using(DisposableType T)
{
  //Do some work with T
}
//T.Dispose() is called automatically
```

A Note on Weak References

Any reference to an object that is on the stack, in a register, in another object, or in one of the other GC Roots will keep an object alive during a GC. This is typically a very good thing, considering that it usually means that your application is not done with that object. There are cases, however, that you want to have a reference to an object, but do not want to affect its lifetime. In these cases the CLR provides a mechanism called Weak References to do just that. Any strong reference—for instance, a reference that roots an object—can be turned into a weak reference. An example of when you might want to use weak references is when you want to create an external cursor object that can traverse a data structure, but should not affect the object's lifetime. Another example is if you wish to create a cache that is flushed when there is memory pressure; for instance, when a GC happens.

Creating a weak reference in C#

```
MyRefType mrt = new MyRefType();
//...

//Create weak reference
WeakReference wr = new WeakReference(mrt);
mrt = null; //object is no longer rooted
//...

//Has object been collected?
if(wr.IsAlive)
{
  //Get a strong reference to the object
  mrt = wr.Target;
  //object is rooted and can be used again
}
else
{
  //recreate the object
  mrt = new MyRefType();
}
```

Managed Code and the CLR JIT

Managed Assemblies, which are the unit of distribution for managed code, contain a processor-independent language called Microsoft Intermediate Language (MSIL or IL). The CLR Just-In-Time (JIT) compiles the IL into optimized native X86 instructions. The JIT is an optimizing compiler, but because compilation happens at runtime, and only the first time a method is called, the number of optimizations that it does needs to be balanced against the time it takes to do the compilation. Typically this is not critical for server applications since startup time and responsiveness is generally not an issue, but it is critical for client applications. Note that startup time can be improved by doing compilation at install time using NGEN.exe.

Many of the optimizations done by the JIT do not have programmatic patterns associated with them, for instance, you cannot explicitly code for them, but there are a number that do. The next section discusses some of those optimizations.

> **HINT** Improve startup time of client applications by compiling your application at install time, using the NGEN.exe utility.

Method Inlining

There is a cost associated with method calls; arguments need to be pushed on the stack or stored in registers, the method prolog and epilog need to be executed and so on. The cost of these calls can be avoided for certain methods by simply moving the method body of the method being called into the body of the caller. This is called Method In-lining. The JIT uses a number of heuristics to decide whether a method should be in-lined. The following is a list of the more significant of those (note that this is not exhaustive):

- Methods that are greater than 32 bytes of IL will not be inlined.
- Virtual functions are not inlined.
- Methods that have complex flow control will not be in-lined. Complex flow control is any flow control other than if/then/else; in this case, switch or while.
- Methods that contain exception-handling blocks are not inlined, though methods that throw exceptions are still candidates for inlining.
- If any of the method's formal arguments are structs, the method will not be inlined.

I would carefully consider explicitly coding for these heuristics because they might change in future versions of the JIT. Don't compromise the correctness of the method to attempt to guarantee that it will be inlined. It is interesting to note that the inline and __inline keywords in

C++ do not guarantee that the compiler will inline a method (though __forceinline does).

Property get and set methods are generally good candidates for inlining, since all they do is typically initialize private data members.

HINT Don't compromise the correctness of a method in an attempt to guarantee inlining.

Range Check Elimination

One of the many benefits of managed code is automatic range checking; every time you access an array using array[index] semantics, the JIT emits a check to make sure that the index is in the bounds of the array. In the context of loops with a large number of iterations and small number of instructions executed per iteration these range checks can be expensive. There are cases when the JIT will detect that these range checks are unnecessary and will eliminate the check from the body of the loop, only checking it once before the loop execution begins. In C# there is a programmatic pattern to ensure that these range checks will be eliminated: explicitly test for the length of the array in the "for" statement. Note that subtle deviations from this pattern will result in the check not being eliminated, and in this case, adding a value to the index.

Range Check Elimination in C#

```
//Range check will be eliminated
for(int i = 0; i < myArray.Length; i++)
{
  Console.WriteLine(myArray[i].ToString());
}

//Range check will NOT be eliminated
for(int i = 0; i < myArray.Length + y; i++)
{
  Console.WriteLine(myArray[i+x].ToString());
}
```

The optimization is particularly noticeable when searching large jagged arrays, for instance, as both the inner and outer loop's range check are eliminated.

Optimizations that Require Variable Usage Tracking

Aa number of JIT compiler optimizations require that the JIT track the usage of formal arguments and local variables; for example, when are they first used and the last time they are used in the body of the method. In version 1.0 and 1.1 of the CLR there is a limitation of 64 on the total number of variables for which the JIT will track usage. An example of an optimization which requires usage tracking is Enregistration. Enregistration is when variables are stored in processor registers as opposed to on the stack frame, for example, in RAM. Access to the Enregistered variables is significantly faster than if they are on the stack frame, even if the variable on the frame happens to be in the processor cache. Only 64 variables will be considered for Enregistration; all other variables will be pushed on the stack. There are other optimizations other than Enregistration that depend on usage tracking. The number of formal arguments and locals for a method should be kept below 64 to ensure the maximum number of JIT optimizations. Keep in mind that this number might change for future versions of the CLR.

HINT Keep methods short. There are a number of reasons for this including method inlining, Enregistration and JIT duration.

Other JIT Optimizations

The JIT compiler does a number of other optimizations: constant and copy propagation, loop invariant hoisting, and several others. There are no explicit programming patterns that you need to use to get these optimizations; they are free.

Why do I not see these optimizations in Visual Studio?

When you use Start from the Debug menu or press F5 to start an application in Visual Studio, whether you have built a Release or Debug version, all the JIT optimizations will be disabled. When a managed application is started by a debugger, even if it is not a Debug build of the application, the JIT will emit non-optimized x86 instructions. If you wish to have the JIT emit optimized code, then start the application from the Windows Explorer or use CTRL+F5 from within Visual Studio. If you would like to view the optimized disassembly and contrast it with the non-optimized code, you can use cordbg.exe.

HINT Use cordbg.exe to see disassembly of both optimized and non-optimized code emitted by the JIT. After starting the application with cordbg.exe, you can set the JIT mode by typing the following:

```
(cordbg) mode JitOptimizations 1
JIT's will produce optimized code

(cordbg) mode JitOptimizations 0
```

JIT's will produce debuggable (non-optimized) code.

Value Types

The CLR exposes two different sets of types, reference types and value types. Reference types are always allocated on the managed heap and are passed by reference (as the name implies). Value types are allocated on the stack or inline as part of an object on the heap, and are passed by value by default, though you can also pass them by reference. Value types are very cheap to allocate and, assuming that they are kept small and simple, they are cheap to pass as arguments. A good example of an appropriate use of value types would be a Point value type that contains an *x* and *y* coordinate.

Point Value Type

```
struct Point
{
  public int x;
  public int y;

  //
}
```

Value types can also be treated as objects; for instance, object methods can be called on them, they can be cast to object, or passed where an object is expected. When this happens however the value type is converted into a reference type, through a process called Boxing. When a value type is Boxed, a new object is allocated on the managed heap and the value is copied into the new object. This is a costly operation and can reduce or entirely negate the performance gained by using value types. When the Boxed type is implicitly or explicitly cast back to a value type, it is Unboxed.

Box/Unbox Value Type

C#:

```
int BoxUnboxValueType()
{
  int i = 10;
  object o = (object)i; //i is Boxed
  return (int)o + 3; //i is Unboxed
}
```

MSIL:

```
.method private hidebysig instance int32
       BoxUnboxValueType() cil managed
{
  // Code size       20 (0x14)
  .maxstack  2
  .locals init (int32 V_0,
           object V_1)
  IL_0000:  ldc.i4.s   10
  IL_0002:  stloc.0
  IL_0003:  ldloc.0
  IL_0004:  box        [mscorlib]System.Int32
  IL_0009:  stloc.1
  IL_000a:  ldloc.1
  IL_000b:  unbox      [mscorlib]System.Int32
  IL_0010:  ldind.i4
  IL_0011:  ldc.i4.3
  IL_0012:  add
  IL_0013:  ret
} // end of method Class1::BoxUnboxValueType
```

If you implement custom value types (struct in C#), you should consider overriding the **ToString** method. If you do not override this method, calls to **ToString** on your value type will cause the type to be Boxed. This is also true for the other methods that are inherited from **System.Object**, in that case, **Equals**, though **ToString** is probably the most often called method. If you would like to know if and when your value type is being Boxed, you can look for the box instruction in the the MSIL using the ildasm.exe utility (as in the snippet above).

Overriding the ToString() method in C# to prevent boxing

```
struct Point
{
  public int x;
  public int y;

  //This will prevent type being boxed when ToString is called
  public override string ToString()
  {
    return x.ToString() + "," + y.ToString();
  }
}
```

Be aware that when creating Collections—for example, an ArrayList of float—every item will be Boxed when added to the collection. You should consider using an array or creating a custom collection class for your value type.

Implicit Boxing when using Collection Classes in C#

```
ArrayList al = new ArrayList();
al.Add(42.0F); //Implicitly Boxed becuase Add() takes object
float f = (float)al[0]; //Unboxed
```

Exception Handling

It is common practice to use error conditions as normal flow control. In this case, when attempting to programmatically add a user to an Active Directory instance, you can simply attempt to add the user and, if an E_ADS_OBJECT_EXISTS HRESULT is returned, you know that they already exist in the directory. Alternately, you could search the directory for the user and then only add the user if the search fails.

This use of errors for normal flow control is a performance anti-pattern in the context of the CLR. Error handling in the CLR is done with structured exception handling. Managed exceptions are very cheap until you throw them. In the CLR, when an exception is thrown, a stack walk is required to find an appropriate exception handler for the thrown exception. Stack walking is an expensive operation. Exceptions should be used as their name implies; in exceptional, or unexpected circumstances.

HINT Consider returning an enumerated result for expected results, as opposed to throwing an exception, for performance-critical methods.

HINT There are a number of .NET CLR Exceptions Performance Counters that will tell you how many exceptions are being thrown in your application.

HINT If you are using VB.NET use exceptions rather than On Error Goto; the error object is an unnecessary cost.

Threading and Synchronization

The CLR exposes rich threading and synchronization features, including the ability to create your own threads, a thread pool, and various synchronization primitives. Before taking advantage of the threading support in the CLR you should carefully consider your use of threads. Keep in mind that adding threads can actually reduce your throughput as opposed to increase it, and you can be sure that it will increase your memory utilization. In server applications that are going to run on multi-processor machines, adding threads can significantly improve throughput by parallelizing execution (though it does depend on how much lock contention is going on, for example, serialization of execution), and in client applications, adding a thread to show activity and/or progress can improve perceived performance (at a small throughput cost).

If the threads in your application are not specialized for a specific task, or have special state associated with them, you should consider using the thread pool. If you have used the Win32 Thread Pool in the past the CLR's Thread Pool will be very familiar to you. There is a single instance of the Thread Pool per managed process. The Thread Pool is smart about the number of threads it creates and will tune itself according to load on the machine.

Threading cannot be discussed without discussing synchronization; all throughput gains that multithreading can give your application can be negated by badly written synchronization logic. The granularity of locks can significantly affect the overall throughput of your application, both because of the cost of creating and managing the lock and the fact that locks can potentially serialize execution. I will use the example of trying to add a node to a tree to illustrate this point. If the tree is going to be a shared data structure, for instance, multiple threads need access to it during the execution of the application, and you will need to synchronize access to the tree. You could choose to lock the entire tree while adding a node, which means you only incur the cost of creating a single lock, but other threads attempting to access the tree will probably block. This would be an example of a coarse-grained lock. Alternatively, you could lock each node as you traverse the tree, which would mean you incur the cost of creating a lock per node, but other threads would not block unless they attempted to access the specific node that you had locked. This is an example of a fined-grained lock. Probably a more appropriate granularity of lock would be to lock only the sub-tree that you are operating on. Note that in this example you would probably use a shared lock (RWLock), because multiple readers should be able to get access at the same time.

The simplest and highest-performance way to do synchronized operations is to use the System.Threading.Interlocked class. The Interlocked class exposes a number of low-level atomic operations: **Increment**, **Decrement**, **Exchange** and **CompareExchange**.

Using the System.Threading.Interlocked Class in C#

```
using System.Threading;
//...
public class MyClass
{
  void MyClass() //Constructor
  {
    //Increment a global instance counter atomically
    Interlocked.Increment(ref MyClassInstanceCounter);
  }

  ~MyClass() //Finalizer
  {
    //Decrement a global instance counter atomically
    Interlocked.Decrement(ref MyClassInstanceCounter);
    //...
  }
  //...
}
```

Probably the most commonly used synchronization mechanism is the Monitor or critical section. A Monitor lock can be used directly or by using the lock keyword in C#. The lock keyword synchronizes access, for the given object, to a specific block of code. A Monitor lock that is fairly lightly contested is relatively cheap from a performance perspective, but becomes more expensive if it is highly contested.

The C# lock keyword

```
//Thread will attempt to obtain the lock
//and block until it does
lock(mySharedObject)
{
  //A thread will only be able to execute the code
  //within this block if it holds the lock
}//Thread releases the lock
```

The RWLock provides a shared locking mechanism: for example, "readers" can share the lock with other "readers", but a "writer" cannot. In the cases that this is applicable, the RWLock can result in better throughput than using a Monitor, which would only allow a single reader or writer to get the lock at a time. The System.Threading namespace also includes the Mutex class. A Mutex is a synchronization primitive that allows for cross-process synchronization. Be aware that this is significantly more expensive than a critical section, and should only be used in the case where cross-process synchronization is required.

Reflection

Reflection is a mechanism provided by the CLR, which allows you to get type information programmatically at runtime. Reflection depends heavily on metadata, which is embedded in managed assemblies. Many reflection APIs require searching and parsing of the metadata, which are expensive operations.

The reflection APIs can be grouped into three performance buckets; type comparison, member enumeration and member invocation. Each of these buckets gets progressively more expensive. Type comparison operations—in this case, **typeof** in C#, **GetType**, **is**, **IsInstanceOfType** and so on—are the cheapest of the reflection APIs, though they are by no means cheap. Member enumerations allow you to programmatically inspect the methods, properties, fields, events, constructors and so on of a class. An example of where these might be used is in design-time scenarios, in this case enumerating properties of Customs Web Controls for the Property Browser in Visual Studio. The most expensive of the reflection APIs are those that allow you to dynamically invoke the members of a class, or dynamically emit, JIT and execute a method. There certainly are late-bound scenarios where dynamic loading of assemblies, instantiations of types, and method invocations are required, but this loose coupling requires an explicit performance tradeoff. In general the reflection APIs should be avoided in performance-sensitive code paths. Note that though you do not directly use reflection, an API that you use might use it. So also be aware of transitive use of the reflection APIs.

Late Binding

Late-bound calls are an example of a feature that uses Reflection under the covers. Visual Basic.NET and JScript.NET both have support for late-bound calls. For instance, you do not have to declare a variable before its use. Late-bound objects are actually of type object and Reflection is used to convert the object to the correct type at runtime. A late-bound call is orders of magnitude slower than a direct call. Unless you specifically need late-bound behavior, you should avoid its use in performance-critical code paths.

HINT If you are using VB.NET and do not explicitly need late binding, you can tell the compiler to disallow it by including the Option Explicit On and Option Strict On at the top of your source files. These options force you to declare and strongly type your variables and turns off implicit casting.

Security

Security is a necessary and integral part of the CLR, and has a performance cost associated with it. In the case that the code is Fully Trusted and the security policy is the default, security should have a minor impact on the throughput and startup time of your application. Partially trusted code—for example, code from the Internet or Intranet Zone—or narrowing the MyComputer Grant Set will increase the performance cost of security.

COM Interop and Platform Invoke

COM Interop and Platform Invoke expose native APIs to managed code in an almost transparent way; calling most native APIs typically requires no special code, though it may require a few mouse clicks. As you might expect, there is a cost associated with calling native code from managed code and vice versa. There are two components to this cost: a fixed cost associated with doing the transitions between native and managed code, and a variable cost associated with any marshalling of arguments and return values that might be required. The fixed contribution to the cost for both COM Interop and P/Invoke is small: typically less than 50 instructions. The cost of marshalling to and from managed types will depend on how different the representations are on either side of the boundary. Types that require a significant amount of transformation will be more expensive. For example, all strings in the CLR are Unicode strings. If you are calling a Win32 API via P/Invoke that expects an ANSI character array, every character in the string has to be narrowed. However if a managed integer array is being passed where a native integer array is expected, no marshalling is required.

Because there is a performance cost associated with calling native code, you should make sure that the cost is justified. If you are going to make a native call, make sure that the work that the native call does justifies the performance cost associated with making the call—keep methods "chunky" rather than "chatty." A good way to measure the cost of a native call is to measure the performance of a native method that takes no arguments and has no return value, and then measure the performance of the native method you wish to call. The difference will give you an indication of the marshalling cost.

> **HINT** Make "Chunky" COM Interop and P/Invoke calls as opposed to "Chatty" calls, and make sure that the cost of making the call is justified by the amount of work that the call does.

Note that there are no threading models associated with managed threads. When you are going to make a COM Interop call, you need to make sure that the thread that the call is going to be made on is initialized to the correct COM threading model. This is typically done using the MTAThreadAttribute and STAThreadAttribute (though it can also be done programmatically).

Performance Counters

A number of Windows performance counters are exposed for the .NET CLR. These performance counters should be a developer's weapon of choice when first diagnosing a performance problem or when attempting to identify the performance characteristics of a managed application. I have already mentioned a few of the counters that relate to memory management and exceptions. There are performance counters for almost every aspect of the CLR and .NET Framework. These performance counters are always available and are non-invasive; they have low overhead and do not change the performance characteristics of your application.

Other Tools

Other than the Performance Counters and the CLR Profiler, you will want to use a conventional profiler to ascertain which methods in your application are taking the most time and being called the most often. These are going to be the methods that you optimize first. A number of commercial profilers are available that support managed code, including DevPartner Studio Professional Edition 7.0 from Compuware and VTune™ Performance Analyzer 7.0 from Intel®. Compuware also produces a free profiler for managed code called DevPartner Profiler Community Edition.

Conclusion

This article just begins the examination of the CLR and the .NET Framework from a performance perspective. There are many other aspects of the architecture of the CLR and the .NET Framework that will affect the performance of your application. The best guidance that I can give to any developer is to not make any assumptions about the performance of the platform your application is targeting and the APIs you are using. Measure everything!

Happy juggling.

Resources

- Compuware DevPartner Studio Professional Edition 7.0.
- Intel.VTune Performance Analyzer 7.0.
- Compuware DevPartner Profiler Community Edition.
- Jan Gray, Writing Faster Code: Knowing What Things Cost, MSDN.
- Rico Mariani, Garbage Collector Basics and Performance Hints, MSDN.