



Writing High Performance .NET Code

Milind P Hanchinmani, Sr. Application Engineer

Introduction

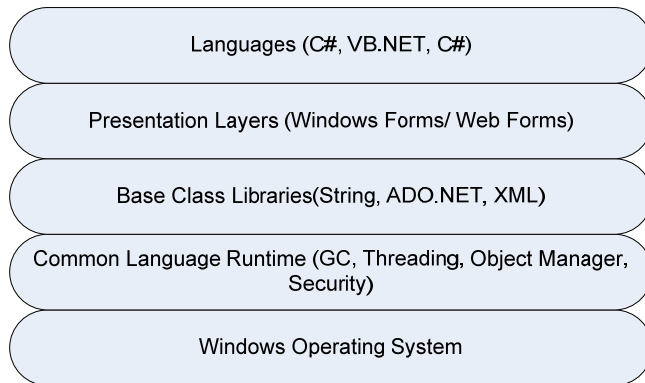
Since most day to day operations are moving online (Core banking, Reservations, Shopping), software performance has become vital to their success. So many times visits to a web site takes long time to load, resulting in frustration and the migration to a different site (similar business). For businesses this can be fatal as they lose customers. Web sites often slow or even go down when traffic increases. Performance/stress testing of your application can help avoid such downtime. There are tools that tell you performance of your application is bad but not necessarily why. But if you have information knowing what to look for, what is good and what is bad, will put your application in better shape later.

With Microsoft® .NET Framework, developers can now build complete business solutions quickly with more functionality and robustness with its rich and easy to use features and functionality. But with this comes increased opportunity for architects and developers to design and build poor, non scalable solutions because architecting and designing these solutions are not really very straight forward. This paper talks about the core performance related issues that one should be aware of in .NET. This paper also talks about some common mistakes which one should avoid and many tips for writing high performance .NET code.

This paper will discuss:

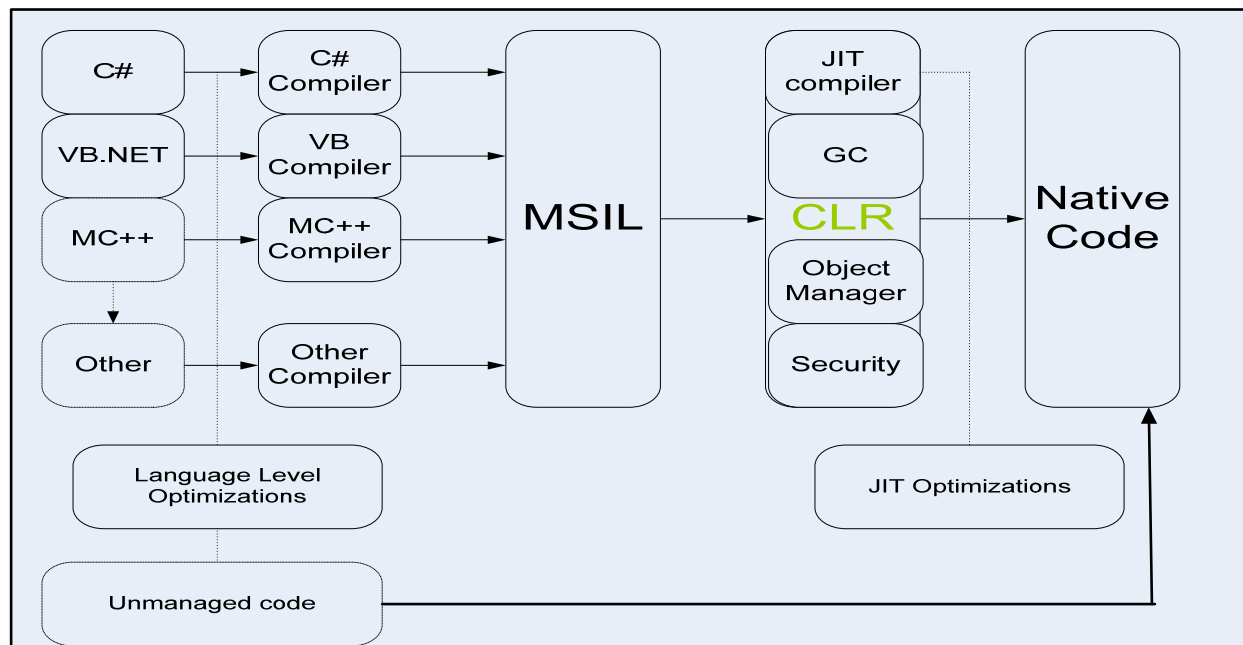
- .NET Framework components and CLR execution Model
- Threading support in .NET and tips for avoiding common threading mistakes
- Automatic Memory management - Writing GC friendly code
- Briefly talks about performance tools available for tuning .NET code

.NET Framework Components and CLR execution Model



The .NET Framework provides a run-time environment called the CLR, which manages the execution of code and provides services that make the development process easier. CLR provides features such as automatic memory management (GC), exception handling, security, type safety, JIT (Just in time compiler for converting msil to native code) and more. CLR is implemented as a dll called "mscorlib.dll". It also has support for Base Class Libraries (BCL) which sits on top of CLR, providing libraries for functionalities such as String, File I/o, and Networking, Collection classes, Data Access (ADO.NET) and XML processing. On top of BCL there are presentation layers (Web Forms and Windows forms), which provide UI functionality. Last, one finds the languages that Microsoft® provides for .NET. Currently there are more than 15 different languages that are targeted for .NET framework.

CLR Execution Model:



Each Language has a compiler which compiles and converts the code to msil (Microsoft® Intermediate Language). There are multiple optimizations that are built into each of these compilers which produce efficient IL code. Then CLR takes over and it has the JIT compiler convert this IL code into native code that CLR can execute. The JIT compiler also has many optimizations built in which can produce efficient native code for better performance. If the code is unmanaged, then we bypass most of this and can directly run unmanaged programs. Note that .NET provides additional features by which we can use pointers to access arrays etc through a feature called "unsafe" for better performance.

Threading support in .NET and tips for avoiding common threading mistakes:

Threading support in .NET is implemented in System.Threading namespace. This provides the classes and functions such as creating/destroying threads, synchronization primitives for atomic access that needed to write multi threaded code. This namespace also provides a class that allows us to use the pool of system provided threads called "ThreadPool".

ThreadPool basically handles thread creation and cleanup. It recycles threads to minimize the thread creation and clean up overhead. ThreadPool also sees other threads running such as GC threads so it can adjust the thread creation logic. A developer may not consider the number of threads that should be used, critical to proper performance. ThreadPool also has built in heuristics enabling it to adjust the number of threads. It is recommended to use thread pool when you are thinking about threading your application. ASP.NET already uses ThreadPool for processing web requests.

I mentioned earlier that ThreadPool automatically decides how many threads are needed for optimal performance. For ASP.NET (web) applications, tune using the *machine.config* file to reduce the contention. Tune using this method when the following conditions are true (2)

- You have available CPU
- Your application performs I/O bound operations and
- The ASP.NET Applications\Requests in Application Queue performance counter indicates that requests are getting queued.

```
<system.web>
```

```
<!--
```

```
<processModel autoConfig="true"/> -> This default means it is adjusted automatically
```

```
-->
```

```
<httpRuntime
```

```
  minFreeThreads="32" -> Requests will be queued if total # of available threads falls below this number.
```

```
  minLocalRequestFreeThreads="32" -> Requests from the local host will be queued if total # of available threads falls below this number.
```

```
/>
```

```
<processModel
```

```
  enable="true"
```

```
  maxWorkerThreads="12" -> maximum # of worker threads in a threadpool. This is per CPU.
```

```
  maxIoThreads="12" -> maximum number of I/O threads in a threadpool. This is per CPU.
```

```
minWorkerThreads="40"      -> minimum worker threads available in the system @ any time. This is
for the entire system
```

```
/>
```

Note: These values are not recommended values but just used for illustration purposes.

So, how does the formula work?

The number of worker threads = $\text{maxWorkerThreads} * \# \text{ of CPU (Cores) in the system} - \text{minFreeThreads}$
 $16 = 12 * 4 - 32$ (assuming you are running a 4 core machine). The total number of concurrent requests you can process is 16. But an interesting question arises. How do you know that this actually worked? Look at the "Pipeline Instance Count" performance counter and it should be equal to 16. Only 1 worker thread can run in a pipeline instance count so you should see a value of 16.

You have to be very careful when doing this as performance may degrade if you use random values.

.NET threading API's and Threadpool make a developer's life easier, but still there are many threading related issues that can hurt performance and scalability.

- Creating more or fewer number of threads than required can impact performance. Use Threadpool to help you in this instance. Ideally, the number of threads will equal the number of cores, and will yield the highest performance as each thread that can run concurrently on a processor.
- Threading wrong portion of application: This is by far the major problem in threading. Analyze your application completely before deciding where to thread. You have to thread the portion of your code where you spend most time to get significant performance.
- Multi threading also complicates debugging and events such as dead locks and race conditions. Have a good debug log (that you can enable in debug mode) to solve some of these complex nature bugs.

Threading Tips:

- a) Distribute the work equally among threads - If the work is imbalanced, one thread will finish the work quickly, but must wait till other thread(s) finish their job, impacting performance.
- b) Don't use too much shared data among threads - If any data or data structure is shared among threads, then synchronization is required to update that data. This increases the amount of serial code/paths in your application hurting your scalability
- c) Acquire Lock late and release it early. This is very important as you must take a lock just before you absolutely have to and release it first before doing anything once the atomic region is executed. Here is an example in .NET

```
void foo ()
{
    int a, b;
    ... //some code
    //Following code has to be atomically executed
    {
    }
    ... //Some other code
    //End of atomic region
}
```

//WRONG: Increased atomic region. Lock will be held longer thus hurting performance

```
void foo ()
{
```

```

    int a, b;
    Object obj ; //for synchronization
    Monitor.Enter(); or lock(obj) {
    ... //some code
    //Following code has to be atomically executed
    {
    }
    ... //Some other code
    Monitor.Exit(); or }
    //End of atomic region
}

//WRONG: Entire function is synchronized. Bad idea.
using System.Runtime.CompilerServices;
MethodImplAttribute(MethodImplOptions.Synchronized)]
void foo ()
{
    int a, b;
    ... //some code
    //Following code has to be atomically executed
    {
    }
    ... //Some other code
    //End of atomic region
}

//Correct: Synchronizing just that block which needs atomic execution
void foo ()
{
    int a, b;
    Object obj;
    ... //some code
    lock(obj) {
        //Following code has to be atomically executed
        {
        }
    } //end of lock
    //End of atomic region
    ... //Some other code
}

```

- d) Use proper synchronization primitives: There are multiple synchronization primitives that are provided by .NET Framework. These vary from fewer features (very fast) to many features (very slow). It is important to use this correctly to get optimal performance. Synchronization primitives can be defined as:
- a. Monitor or lock: Provides a mechanism that synchronizes access to objects
 - b. Interlocked: Provides atomic access to variables that are shared by multiple threads. For example: for any atomic ++ or -- operations consider using Interlocked class
 - c. Mutex: Synchronization primitives that can be used for inter process synchronization. They are considerably slower; use it when you absolutely need it.

- d. ReaderWriterLock: Lock that supports single writer and multiple readers. If you have a scenario where you read your data frequently but update only once in a while, consider using this as it supports multiple readers.
 - e. ReaderWriterLockSlim: Similar to ReaderWriterLock but simplified rules for recursion and for upgrading and downgrading lock state. It also avoids many cases of potential deadlock and has improved performance. Using this is recommended.
 - f. Semaphore: Limits # of threads that can access a resource or pool of resources concurrently. Use it only when you need to control pool of resources.
- e) Never use Thread.Suspend and Thread.Resume to synchronize activities. The suspend and resume operation doesn't happen immediately as CLR has to make sure the execution control is in safe point. This can lead to race conditions or deadlock (1)
 - f) Never use Thread.Abort to abort another thread: (1)
 - g) Don't lock "this" and "type" of an object: Locking **this** pointer is a bad idea as this can have correctness issue as it is visible. Similarly, locking type of an object is a bad idea as these objects are the same across application domains and so thus we lock all instances of objects across app domains in a process.

```
//Wrong
lock (this) {
    do something;
}

//Correct
public class foo {
    Object sync_obj = new Object();
    lock(sync_obj) {
        Do something
    }
}
```

```
//Wrong
lock(typeof(foo))
{
    Do something;
}

//Correct
public class foo {
    private static Object sync_obj = new Object();
    lock (sync_obj) {
        Do something;
    }
}
```

- h) Consider using [ThreadStatic] to eliminate or reduce lock contention: If you can have a data as part of thread local storage (per thread) rather than sharing and after the threads have completed the jobs, you can process the combined effect. Consider using this to reduce lock contention.
- i) Acquire and release lock in the same order: Otherwise you can cause deadlock condition

```
Thread1
lock(obj_A) {
lock(obj_B) {
    Do something;
}
}

Thread2
lock(obj_B) {
    lock(obj_A) {
        Do something ;
    }
}
```

- j) All collections in .NET are not thread safe. Some collection classes (ex: ArrayList) allow multiple readers concurrently. Need to call "Synchronized" method for making it thread safe for updates

```
ArrayList myAr = new ArrayList();

ArrayList mySyncAr = ArrayList.Synchronized (myAr); //use
mySyncAr
```

- k) Enumerating through collections is also “not” thread safe even though it is synchronized. If another thread modifies the underlying collection then an exception will be thrown.

Automatic Memory Management (Garbage Collection)

Automatic memory management, aka GC is one of the most important features provided by .NET Framework. GC manages the allocation and reclaiming of memory in your application. When ever you call “new” to create a new object, GC will allocate memory from managed heap as long as space is available and once it runs out of memory it triggers collection, reclaim memory so that it can start allocating again. We will go into some detail about GC algorithms, how they work, different GC flavors, and how you can write a GC friendly code.

.NET GC is a generational and mark and compact algorithm. We have 3 generations (Gen0, 1 and 2). .NET GC assumes that most of the objects you create die young, so only a part of your entire managed heap can be collected (which is much faster) than collecting the entire managed heap. GC first marks the root objects (to find out those who are alive) and then compacts the heap (moving all live objects to a part of the heap which forms older generation(s). Always, allocations happen in Gen0 heap. The initial gen0 heap is some fraction of the last level cache. The idea is to have gen0 fit in the cache to avoid cache misses.

.NET GC Flavors:

- Workstation GC (WKS)
- Server GC (SVR)

Note: Selecting appropriate GC flavor is essential for optimal performance of your application

Workstation (WKS) GC: WKS GC has 2 variants. Concurrent GC (on) which is the default and can be turned off. Concurrent GC (on) will have less pause time, increasing the UI responsiveness. GC stops the application threads for a shorter duration when absolutely necessary. If you have a throughput kind of application (console app non UI) then turning off concurrent GC might get you better performance. In your application configuration file (ex: foo.exe.config), you can add following [2]

```
<configuration>
```

```
<runtime>
```

```
<gcConcurrent enabled="false"/>
```

```
</runtime>
```

```
</configuration>
```

WKS GC has 1 heap per process and it has 1 GC thread per process. WKS GC is the default even on multiprocessor systems for any non ASP.NET application. ASP.NET automatically chooses SVR GC if you are on a multi processor system.

Server (SVR) GC: As the name suggests, SVR GC is optimized for server based applications (better scalability). It has 1 GC heap per Processor and 1 GC thread per 1 GC heap. For example, if you are on a 4 processor system, you will have 4 heaps and 4 GC threads operating on each of those heaps. A process can create objects in multiple heaps (for load balancing the allocation on heaps) and as mentioned above it is not the default. To enable Server GC, add the following in application configuration files.

```
<configuration> [2]
```

```
<runtime>
```

```
<gcServer enabled="true"/>
```

</runtime>

</configuration>

Tips for selecting appropriate GC:

- For all server throughput related applications, consider selecting Server GC
- ASP.NET web applications on >1 proc machine automatically selects SVR GC. However if you want to run web garden scenario, then consider using WKS GC as the memory foot print might be really high as a result of multiple w3wp processes. SVR GC assumes it is the king and so will try to grab as many resources as possible. So if you have multiple processes running SVR GC, there can be degradation in performance and also an increase usage of system resources. In order to enable ASP.NET using WKS GC, add the following configuration to the **Aspnet.config** file. This is in the same directory as Aspnet_isapi.dll.

```
<configuration>
  <runtime>
    <gcServer enabled="false" />
    <gcConcurrent enabled="false" /> </runtime>
  </configuration>
```
- If you have a client - UI application which requires UI responsiveness, consider selecting WKS GC with Concurrent enabled
- If you have Console application (no UI but throughput app) then consider turning off concurrent GC for better performance.
- If you want lesser resource utilization in a system (memory etc) then consider using WKS GC.

Note: When you ask for Server GC on a UP machine, you get WKS GC with concurrent off. CLR assumes that since you are asking SVR GC, you are more interested in throughput than UI responsiveness and so automatically turn off concurrent GC.

Tips for writing GC Friendly code:

- a. Never Call GC.Collect from your code: .NET GC is a dynamically tuning GC. At every collection it collects information such as survivor rate, and tunes its internal GC tuning parameters so the next GC is more effective than its previous GC. Unlike Java, .NET GC doesn't expose many tuning parameters for the developer. So when you call GC.Collect in your code, it collects those parameters. Since you induced GC the next GC will not be as productive. Also, if GC.Collect is executing not just once but many times (lets say before you start expensive time consuming work and so you need more memory) then GC will not be productive at all. But there is an exception. If you know that you opened a custom form and made some configuration changes and you know that you are not going to need that form any time sooner, you can go ahead and call GC.Collect() so all the long live objects in Gen2 are now dead. It is recommended to use the following code (starting from Orcas build). Here even though GC.Collect is called on gen2, GC will decide if it is helpful if it collects (2nd parameter - Optimized). This is not available in VS2005 and older versions.

```
using System;
class Program
{
    static void Main(String[] args) {
        GC.Collect(2, GCCollectionMode.Optimized);
    }
}
```

}

- b. Create objects that die young: .NET GC is optimized on the premises that most of objects allocated are temporary and die young so they can be collected in gen0 which is cheap. (2)
- c. Don't allocate too many objects: One little line of code could trigger a lot of allocations. Most of the time, it is an allocation that triggers a collection. Keep an eye on what you allocate particularly in loops (2)
- d. Don't allocate too many almost long-life objects: Objects that are neither temporary nor long lived end up in Gen2 and die. This puts pressure on gen2 heap and you may end up doing full collections which is expensive.(2)
- e. Don't allocate too many temporary large objects: Large objects (>85K size) are allocated on a separate large objects heap which is never compacted (it is expensive to move many large objects during compaction). This could put pressure on large object heap, resulting in your doing full collections, also expensive.(2)
- f. Dispose and Finalize: Implement these only when needed. Make sure you call these when an exception occurs (to avoid a memory leak). Also make sure you implement finalize only when you have an unmanaged resource and keep the code very simple. (2)

Tips for improving manage code performance:

We covered threading and GC and now we cover the general VM, code generation and basic ASP.NET and ADO.NET tips for writing better code

- a. Avoid unnecessary boxing [1]

```
int i = 123;
object o = i; (Implicit boxing) //box keyword
int j = (int)o; //unbox keyword
```

When ever we box, a new object is created on the managed heap and the value is copied in it. If we are doing this frequently, then we will create lot of objects (affect GC) and also the extra code we execute for boxing and unboxing.

- b. Consider using strong typed arrays or generics (Visual Studio 2005 onwards)

```
Foo myFoo = new Foo();
myArrayList.Add(myFoo);
Foo myFoo = (Foo) myArrayList[i]; //castclass keyword
```

Collection classes take generic "object" as a parameter. Type casting is required when retrieving objects(your type) from the collection classes. This requires an expensive run time type check by looking at method table of that object. If your object is inherited then this may require traversing one level up which is again expensive. You can avoid this by using generics (similar to C++ template) as shown below which doesn't require run time type check as it is known at the compile time.

```
List<Foo> myList = new List<Foo>();
Foo myfoo = myList[i]; //no check reqd
```

- c. Throw fewer exceptions: Throwing exceptions can be expensive as stack walk is required etc for managing the frames. Don't use exceptions as a control flow in your application

<Wrong>

```

void foo (int parameter)
{
    int ret = 0;
    val = ... ;
    try
        {
            ret = val / parameter;
        }catch(DivideByZeroException) { return ERROR_VAL ;}
}

```

<Correct>

```

void foo (int parameter)
{
    if (parameter == 0) return ERROR_VAL; else {... ;}
}

```

- d. Use StringBuilder for complex string manipulation: Whenever you modify a string (such as append etc), it will create a new string leaving the first one to be collected. Consider using a StringBuilder if you have > 5-7 string manipulations.
- e. Don't use too many Reflection API's: Reflection API's depend on the metadata embedded in assemblies. Thus parsing and searching this information is very expensive.
- f. Don't make functions unnecessarily virtual or synchronized: JIT might disable some optimizations and so the generated code might not be optimal
- g. Don't write big functions: JIT might disable optimizations for faster compile (JIT) time.
- h. Avoid calling small functions inside loop: Consider inlining yourself (incase JIT has not done it). Any mistake in the loop is magnified.
- i. Prefer arrays to collections unless you need that additional functionality that collection classes provide [1]
- j. Use jagged arrays instead of multi dimensional arrays since the former has some special MSIL optimizations for faster array access [1]
- k. Smaller working set produces better performance and so consider using ngen for shared pages
- l. Don't make too many Pinvoke calls (chatty calls) and do less work in unmanaged code: The overhead of transitions (managed to unmanaged and back) can negate performance speedup or even hurt.

Ngen: Ngen.exe (shipped with CLR) invokes JIT compiler on MSIL to create native code and stores it in the disk. Once the native image is created, runtime uses this image automatically each times it runs the assembly. Using native image will eliminate compiling on the fly using JIT compiler at runtime thus reducing application startup time.

Ngen.exe can help improving application performance by,

- Reducing the application startup time - Consider using ngen.exe for improving startup time of your winform based application. Always measure with and without ngening of your application.

- By reducing the total memory consumed by application that use shared assemblies (which are loaded in to different application domains)

Interop: When you build applications in managed code, some times it is necessary to call unmanaged libraries such as calling a COM component. In some cases, you want to use unmanaged code for some performance related reasons as well (such as calling 3rd party highly optimized libraries). CLR provides several ways to do this.

- Using Pinvoke (Platform Invoke) – Allows calling of Windows DLL's, Win32 API's or custom dll's from managed code (1)
- Using MC++ (IJW) – For users for MC++ to call standard DLL's (1)
- COM Interop – Manage languages to call COM components through COM interfaces. (1)

Improving Interop performance: (1)

- Avoid chatty calls that increase unnecessary round trips: Increases the overhead due to multiple transitions
- Avoid inefficient marshalling of parameters: this causes unnecessary waste of system processing cycles
- Properly cleanup (dispose) unmanaged components: This can affect server's memory utilization and can cause memory leaks
- Don't aggressively pin the short lived objects: This can create fragmentation in managed heap hurting performance.

Improving ASP.NET Performance:

- a. Use efficient caching strategies (1): A well designed caching strategies is the single most important consideration during design phase of your application. There are different caching methods such as output caching, partial page caching etc can reduce round trips to database. It is very important to do analysis to figure out where caching is appropriate. You want to consider caching if,
 - a. The data or output is very expensive to create or fetch
 - b. Frequency of use
 - c. The data is fairly static and is not changing frequently.
- b. Partition your application logically: Presentation, business logic, ado.net (database) layer so it is easy to maintain and optimize individual layers (1)
- c. Consider disabling IIS logging on application server
- d. Use server controls efficiently: This can increase the page load time.
- e. Improve page response times: Use Page.IsPostBack to minimize the round trip to server
- f. Ensure pages are batch compiled: If we mix different languages in the same directory then it won't compile all the pages into 1 assembly.
- g. Ensure the debug attribute is not set on pages
- h. Validate and fail early to avoid expensive work
- i. Use view state only when necessary – View state is serialized and deserialized on the server which is expensive

Improving ADO.NET performance:

- a. Use stored procedures – easy for maintenance and for improving SQL performance
- b. Analyze and use data reader and data set appropriately (1). Consider using data readers if,
 - i. You don't need to cache the data or data is read-only
 - ii. When you want to fetch many records rapidly
 - iii. If you don't have to chose random records
- c. Use try{} and finally{} to make sure the connections are closed: If an exception occurs and you have written close at the end, try block or in catch block; it might not execute and keep these connections open
- d. Get only the data that you need from the database: To minimize the time
- e. Use appropriate type of transaction (SQL, ADO.NET and ASP.NET level) and minimize the transaction duration
- f. Use paging mechanism if you want to get large data sets from database for better user experience and to reduce the time

Till now, we have seen tips, tricks and BKM's for writing high performance .NET code. What follows is a brief list of performance tools that are available for tuning .NET code. This paper will not detail them.

Perfmon – System level tool. It exposes several CLR, ASP.NET related counters and this should be used as the first tool for analyzing any .Net applications. I will go in to detail on the counters available and some tips in later posts.

Intel® Vtune™ Analyzer: Profiling tool from Intel which supports .NET including ASP.Net applications.

CLR Profiler: Tool from Microsoft which is used to profile memory (allocation) of your application. It is free and downloadable from msdn.

SOS: Manage debugging extensions from Microsoft. Free, Shipped as SOS.dll with CLR. Exposes many CLR internal data structures such as GC, Exceptions, Objects, Locking etc. Can be used to identify functionality bugs (such as OutOfMemoryException) and performance related bugs as well (locking etc).

VSTS Profiler: A built in profiler from Microsoft® Visual Studio Team system 2008. Can sample application and identify hotspots and hot call chains etc

VSTS: The Microsoft® Visual Studio Team system (for testers) has a built in ability to do performance load testing of n-tier web based applications. It is very simple to use including a recording facility for URL's and also has ability to look @ perfmon counters of all the machines from a client system etc.

[Summary]

In this new Internet era application performance is essential to be successful and to stay ahead of the competition. Including performance engineering throughout the SDLC (software development life cycle) is essential to achieve/exceed performance goals. Performance engineering should be proactive and not reactive (example: When customer complains of a problem). This paper outlines information, tips and BKM's for improving performance and looking at potential issues in threading etc if you are developing your application using Microsoft® Framework SDK.

[Author Bio and Photo]



Milind is currently a Senior Application Engineer with 13 years of industry experience with over 7+ years @ Intel. He joined APAC enabling team last year and has been working with Enterprise ISV's and SI's to improve their application performance on Intel® Architecture. At Intel, in his previous role for 6 years Milind worked as a member of the Intel on-site team at Microsoft in driving improvements in the quality and performance of three generations of Microsoft® .NET Framework for Intel Architecture. His expertise includes performance methodologies, benchmarking and .NET CLR internals.

[References]

- 1) <http://msdn.microsoft.com/en-us/library/ms998530.aspx>
- 2) <http://blogs.msdn.com/maoni>

Note to Intel authors:

Juan A Rodriguez, Intel Corporation
Simonijt Dutta (Intel Corporation)