Performance Considerations for Run-Time Technologies in the .NET Framework

A .NET Developer Platform White Paper

Emmanuel Schanzer
Microsoft Corporation

August 2001

Summary: This article includes a survey of various technologies at work in the managed world and a technical explanation of how they impact performance. Learn about the workings of garbage collection, the JIT, remoting, ValueTypes, security and more. (27 printed pages)

Contents

Overview

The .NET run time introduces several advanced technologies aimed at security, ease-of-development and performance. As a developer, it is important to understand each of the technologies and use them effectively in your code. The advanced tools provided by the run time make it easy to build a robust application, but making that application fly fast is (and always has been) the responsibility of the developer.

This white paper should provide you will a deeper understanding of the technologies at work in .NET, and help you tune your code for speed. Note: this is not a spec sheet. There is plenty of solid technical information out there already. The goal here is to provide the information with a strong tilt towards performance, and may not answer every technical question you have. I recommend looking further in the MSDN Online Library if you don't find the answers you seek here.

I'm going to cover of the following technologies, providing a high-level overview of their purpose and why they affect performance. Then I'll delve into some lower-level implementation details and use sample code to illustrate the ways to get speed out of each technology.

Garbage Collection

The Basics

Garbage collection (GC) frees the programmer from common and difficult-to-debug errors by freeing memory for objects that are no longer used. The general path followed for an object's lifetime is as follows, in both managed and native code:

```
Foo a = new Foo();          // Allocate memory for the object and
    initialize
?a?                    // Use the object
delete a;                // Tear down the state of the object, clean up
                         // and free the memory for that object
```

In native code, you need to do all these things yourself. Missing the allocation or cleanup phases can result in totally unpredictable behavior that is difficult to debug, and forgetting to free objects can result in memory leaks. The path for memory allocation in the Common Language Runtime (CLR) is very close to the path we've just covered. If we add the GC-specific information, we end up with something that looks very similar.

```
Foo a = new Foo();       // Allocate memory for the object and
    initialize
?a?                   // Use the object (it is strongly reachable)
a = null;             // A becomes unreachable (out of scope, nulled,
    etc)
                      // Eventually a collection occurs, and a's
    resources
                      // are torn down and the memory is freed
```

Until the object can be freed, the same steps are taken in both worlds. In native code, you need to remember to free the object when you're done with it. In managed code, once the object is no longer reachable, the GC can collect it. Of course, if your resource requires special attention to be freed (say, closing a socket) the GC may need help to close it correctly. The code you've written before to clean up a resource before freeing it still applies, in the form of **Dispose()** and **Finalize()** methods. I'll talk about the differences between these two later on.

If you keep a pointer to a resource around, the GC has no way of knowing if you intend to use it in the future. What this means is that all of the rules you've used in native code for explicitly freeing objects still apply, but most of the time the GC will handle everything for you. Instead of worrying about memory management one hundred percent of the time, you only have to worry about it about five percent of the time.

The CLR Garbage Collector is a generational, mark-and-compact collector. It follows several principles that allow it to achieve excellent performance. First, there is the notion that objects that are short-lived tend to be smaller and are accessed often. The GC divides the allocation graph into several sub-graphs, called *generations*, which allow it to spend as little time collecting as possible. Gen 0 contains young, frequently used objects. This also tends to be the smallest, and takes about 10 milliseconds to collect. Since the GC can ignore the other generations during this collection, it provides much higher performance. G1 and G2 are for larger, older objects and are collected less frequently. When a G1 collection occurs, G0 is also collected. A G2 collection is a full collection, and is the only time the GC traverses the entire graph. It also makes intelligent use of the CPU caches, which can tune the memory subsystem for the specific processor on which it runs. This is an optimization not easily available in native allocation, and can help your application improve performance.

When Does a Collection Happen?

When a time allocation is made, the GC checks to see if a collection is needed. It looks at the size of the collection, the amount of memory remaining, and the sizes of each generation, and then uses a heuristic to make the decision. Until a collection occurs, the speed of object allocation is usually as fast (or faster) than C or C++.

What Happens When a Collection Occurs?

Let’s walk through the steps a garbage collector takes during a collection. The GC maintains a list of roots, which point into the GC heap. If an object is live, there is a root to its location in the heap. Objects in the heap can also point to each other. This graph of pointers is what the GC must search through to free up space. The order of events is as follows:

1. The managed heap keeps all of its allocation space in a contiguous block, and when this block is smaller than the amount requested, the GC is called.
2. The GC follows each root and all the pointers that follow, maintaining a list of the objects that are *not* reachable.
3. Every object not reachable from any root is considered collectable, and is marked for collection.

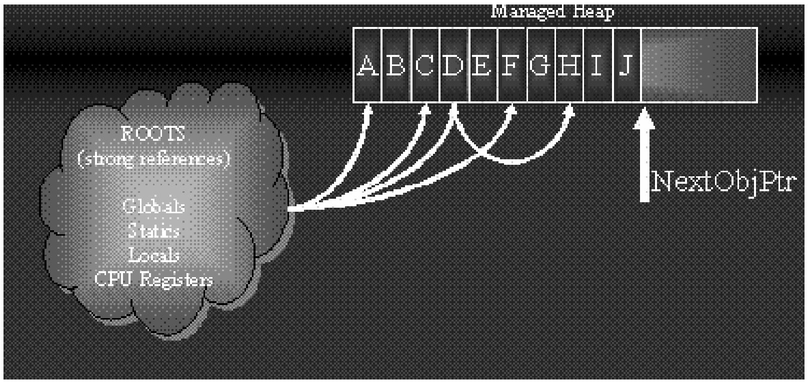

Figure 1. Before Collection: Note that not all blocks are reachable from roots!

4. Removing objects from the reachability graph makes most objects collectable. However, some resources need to be handled specially. When you define an object, you have the option of writing a **Dispose()** method or a **Finalize()** method (or both). I’ll talk about the differences between the two, and when to use them later.
5. The final step in a collection is the compaction phase. All the objects that are in use are moved into a contiguous block, and all pointers and roots are updated.
6. By compacting the live objects and updating the start address of the free space, the GC

maintains that all free space is contiguous. If there is enough space to allocate the object, the GC returns control to the program. If not, it raises an `OutOfMemoryException`.

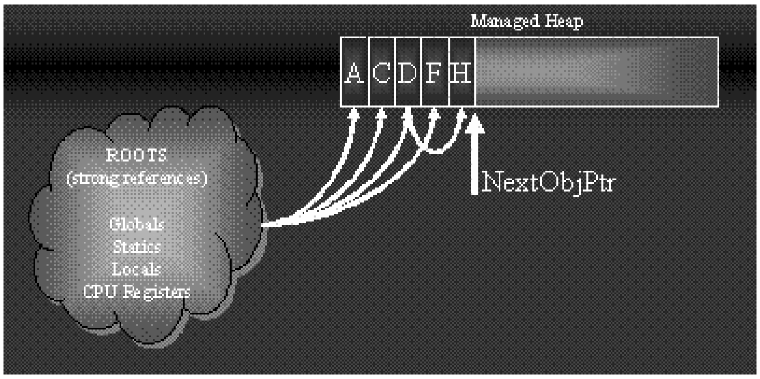

Figure 2. After Collection: The reachable blocks have been compacted. More free space!

For more technical information on memory management, see Chapter 3 of *Programming Applications for Microsoft Windows* by Jeffrey Richter (Microsoft Press, 1999).

Object Cleanup

Some objects require special handling before their resources can be returned. A few examples of such resources are files, network sockets, or database connections. Simply releasing the memory on the heap isn't going to be enough, since you want these resources closed gracefully. To perform object cleanup, you can write a **Dispose()** method, a **Finalize()** method, or both.

A **Finalize()** method:

- Is called by the GC
- Is not guaranteed to be called in any order, or at a predictable time
- After being called, frees memory after the *next* GC
- Keeps all child objects live until the next GC

A **Dispose()** method:

- Is called by the programmer
- Is ordered and scheduled by the programmer
- Returns resources upon completion of the method

Managed objects that hold only managed resources don't require these methods. Your program will probably use only a few complex resources, and chances are you know what they are and when you need them. If you know both of these things, there's no reason to rely on finalizers, since you can do the cleanup manually. There are several reasons that you want to do this, and they all have to do with the *finalizer queue*.

In the GC, when an object that has a finalizer is marked collectable, it and any objects it points to are placed in a special queue. A separate thread walks down this queue, calling the **Finalize()** method of each item in the queue. *The programmer has no control over this thread, or the order of items placed in the queue.* The GC may return control to the program, without having finalized any objects in the queue. Those objects may remain in memory, tucked away in queue for a long time. Calls to finalize are done automatically, and there is no direct performance impact from call itself. However, the non-deterministic model for finalization can *definitely* have other indirect consequences:

- In a scenario where you have resources that need to be released at a specific time, you lose control with finalizers. Say you have a file open, and it needs to be closed for security reasons. Even when you set the object to null, and force a GC immediately, the file will remain open until its **Finalize()** method is called, and you have no idea when this could happen.
- N objects that require disposal in a certain order may not be handled correctly.
- An enormous object and its children may take up far too much memory, require additional collections and hurt performance. These objects may not be collected for a long time.
- A small object to be finalized may have pointers to large resources that could be freed at any time. These objects will not be freed until the object to be finalized is taken care of, creating unnecessary memory pressure and forcing frequent collections.

The state diagram in Figure 3 illustrates the different paths your object can take in terms of finalization or disposal.

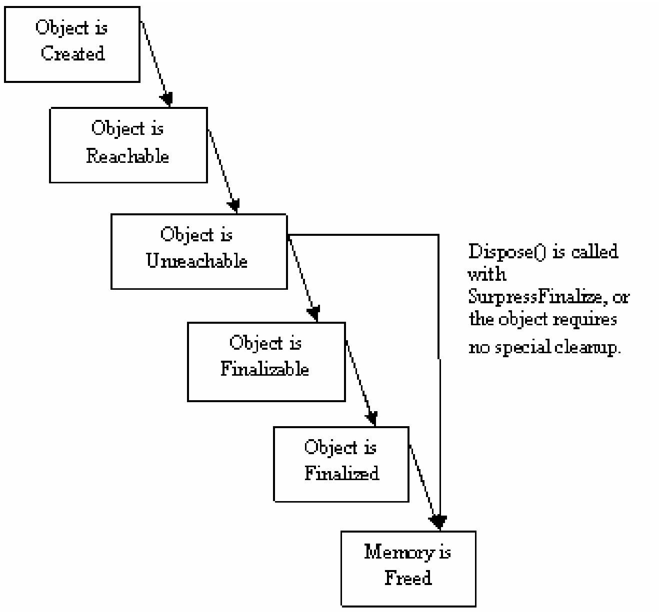

Figure 3. Disposal and finalization paths an object can take

As you can see, finalization adds several steps to the object's lifetime. If you dispose of an object yourself, the object can be collected and the memory returned to you in the next GC. When finalization needs to occur, you have to wait until the actual method gets called. Since you are not given any guarantees about when this happens, you can have a lot of memory tied up and be at the mercy of the finalization queue. This can be extremely problematic if your object is connected to a whole tree of objects, and they all sit in memory until finalization occurs.

Choosing Which Garbage Collector to Use

The CLR has two different GCs: Workstation (mscorwks.dll) and Server (mscorsvr.dll). When running in Workstation mode, latency is more of a concern than space or efficiency. A server with multiple processors and clients connected over a network can afford some latency, but throughput is now a top priority. Rather than shoehorn both of these scenarios into a single GC scheme, Microsoft has included two garbage collectors that are tailored to each situation.

Server GC:

- Multiprocessor (MP) Scalable, Parallel
- One GC thread per CPU
- Program paused during marking

Workstation GC:

- Minimizes pauses by running concurrently during full collections

The server GC is designed for maximum throughput, and scales with very high performance. Memory fragmentation on servers is a much more severe problem than on workstations, making garbage collection an attractive proposition. In a uniprocessor scenario, both collectors work the same way: workstation mode, without concurrent collection. On an MP machine, the Workstation GC uses the second processor to run the collection concurrently, minimizing delays while diminishing throughput. The Server GC uses multiple heaps and collection threads to maximize throughput and scale better.

You can choose which GC to use when you host the run time. When you load the run time into a process, you specify what collector to use. Loading the API is discussed in the .NET Framework Developer's Guide. For an example of a simple program that hosts the run time and selects the server GC, take a look at the Appendix.

Myth: Garbage Collection Is Always Slower Than Doing It by Hand

Actually, until a collection is called, the GC is a lot faster than doing it by hand in C. This surprises a lot of people, so it's worth some explanation. First of all, notice that finding free space occurs in constant time. Since all free space is contiguous, the GC simply follows the pointer and checks to see if there's enough room. In C, a call to **malloc()** typically results in a search of a linked list of free blocks. This can be time consuming, especially if your heap is badly fragmented. To make matters worse, several implementations of the C run time lock the heap during this procedure. Once the memory is allocated or used, the list has to be updated. In a garbage-collected environment, allocation is free, and the memory is released during collection. More advanced programmers will reserve large blocks of memory, and handle allocation within that block themselves. The problem with this approach is that memory fragmentation becomes a huge problem for programmers, and it forces them to add a lot of memory-handling logic to their applications. In the end, a garbage collector doesn't add a lot of overhead. Allocation is as fast or faster, and compaction is handled automatically—freeing programmers to focus on their applications.

In the future, garbage collectors could perform other optimizations that make it even faster. Hot spot identification and better cache usage are possible, and can make enormous speed differences. A smarter GC could pack pages more efficiently, thereby minimizing the number of page fetches that occur during execution. All of these could make a garbage-collected environment faster than doing things by hand.

Some people may wonder why GC isn't available in other environments, like C or C++. The answer is types. Those languages allow casting of pointers to any type, making it extremely difficult to know what a pointer refers to. In a managed environment like the CLR, we can guarantee enough about the pointers to make GC possible. The managed world is also the only place where we can safely stop thread execution to perform a GC: in C++ these operations are either unsafe or very limited.

Tuning for Speed

The biggest worry for a program in the managed world is *memory retention*. Some of the problems that you'll find in unmanaged environments are not an issue in the managed world: memory leaks and dangling pointers are not much of a problem here. Instead, programmers need to be careful about leaving resources connected when they no longer need them.

The most important heuristic for performance is also the easiest one to learn for programmers who are used to writing native code: keep track of the allocations to make, and free them when you're done. The GC has no way of knowing that you aren't going to use a 20KB string that you built if it's part of an object that's being kept around. Suppose you have this object tucked away in a vector somewhere, and you never intend to use that string again. Setting the field to null will let the GC collect those 20KB later, even if you still need the object for other purposes. If you don't need the object anymore, make sure you're not keeping references to it. (Just like in native code.) For smaller objects, this is less of a problem. Any programmer that's familiar with memory management in native code will have no problem here: all the same common sense rules apply. You just don't have to be so paranoid about them.

The second important performance concern deals with object cleanup. As I mentioned earlier, finalization has profound impacts on performance. The most common example is that of a managed handler to an unmanaged resource: you need to implement some kind of cleanup method, and this is where performance becomes an issue. If you depend on finalization, you open yourself up to the performance problems I listed earlier. Something else to keep in mind is that the GC is largely unaware of memory pressure in the native world, so you may be using a ton of unmanaged resources just by keeping a pointer around in the managed heap. A single pointer doesn't take up a lot of memory, so it could be a while before a collection is needed. To get around these performance problems, while still playing it safe when it comes to memory retention, you should pick a design pattern to work with for all the objects that require special cleanup.

The programmer has four options when dealing with object cleanup:

1. Implement Both

   This is the recommended design for object cleanup. This is an object with some mix of unmanaged and managed resources. An example would be **System.Windows.Forms.Control**. This has an unmanaged resource (HWND) and potentially managed resources (DataConnection, etc.). If you are unsure of when you make use of unmanaged resources, you can open the manifest for your program in `ILDASM` and check for references to native libraries. Another alternative is to use `vadump.exe` to see what resources are loaded along with your program. Both of these may provide you with insight as to what kind of native resources you use.

The pattern below gives users a single recommended way instead of overriding cleanup logic (override **Dispose(bool)**). This provides maximum flexibility, as well as catch-all just in case **Dispose()** is never called. The combination of maximum speed and flexibility, as well as the safety-net approach make this the best design to use.

Example:

```
public class MyClass : IDisposable {
  public void Dispose() {
    Dispose(true);
    GC.SuppressFinalizer(this);
  }
  protected virtual void Dispose(bool disposing) {
    if (disposing) {
      ?
    }
      ?
  }
  ~MyClass() {
    Dispose(false);
  }
}
```

2. Implement **Dispose()** Only

   This is when an object has only managed resources, and you want to make sure that its cleanup is deterministic. An example of such an object is **System.Web.UI.Control**.

   Example:

```
public class MyClass : IDisposable {
  public virtual void Dispose() {
    ?
  }
```

3. Implement **Finalize()** Only

   This is needed in extremely rare situations, and I strongly recommend against it. The implication of a **Finalize()** only object is that the programmer has no idea when the object is going to be collected, yet is using a resource complex enough to require special cleanup. This situation should never occur in a well-designed project, and if you find yourself in it you should go back and find out what went wrong.

   Example:

```
public class MyClass {
  ?
  ~MyClass() {
```

```
        ?
    }
```

4. Implement Neither

   This is for a managed object that points only to other managed objects that are not disposable nor to be finalized.

Recommendation

The recommendations for dealing with memory management should be familiar: release objects when you're done with them, and keep an eye out for leaving pointers to objects. When it comes to object cleanup, implement both a **Finalize()** and **Dispose()** method for objects with unmanaged resources. This will prevent unexpected behavior later, and enforce good programming practices

The downside here is that you force people to have to call **Dispose()**. There is no performance loss here, but some people might find it frustrating to have to think about disposing of their objects. However, I think it's worth the aggravation to use a model that makes sense. Besides, this forces people to be more attentive to the objects they allocate, since they can't blindly trust the GC to always take care of them. For programmers coming from a C or C++ background, forcing a call to **Dispose()** will probably be beneficial, since it's the kind of thing they are more familiar with.

Dispose() should be supported on objects that hold on to unmanaged resources anywhere in the tree of objects underneath it; however, **Finalize()** need only be placed only on those objects that are specifically holding on to these resources, such as an OS Handle or unmanaged memory allocation. I suggest creating small managed objects as "wrappers" for implementing **Finalize()** in addition to supporting **Dispose()**, which would be called by the parent object's **Dispose()**. Since the parent objects do not have a finalizer, the entire tree of objects will not survive a collection regardless of whether or not **Dispose()** was called.

A good rule of thumb for finalizers is to use them only on the most primitive object that *requires* finalization. Suppose I have a large managed resource that includes a database connection: I would make it possible for the connection itself to be finalized, but make the rest of the object disposable. That way I can call **Dispose()** and free the managed portions of the object immediately, without having to wait for the connection to be finalized. Remember: use **Finalize()** only *where* you have to, *when* you have to.

> **Note** C and C++ Programmers: the Destructor semantic in C# creates a *finalizer*, not a disposal method!

Thread Pool

The Basics

The thread pool of the CLR is similar to the NT thread pool in many ways, and requires almost no new understanding on the part of the programmer. It has a wait thread, which can handle the blocks for other threads and notify them when they need to return, freeing them to do other work. It can spawn new threads and block others to optimize for CPU utilization at run time, guaranteeing that the greatest amount of useful work is done. It also recycles threads when they are done, starting them up again without the overhead of killing and spawning new ones. This is a substantial performance boost over handling threads manually, but it is not a catch-all. Knowing when to use the thread pool is essential when tuning a threaded application.

What you know from the NT thread pool:

- The thread pool will handling thread creation and cleanup.
- It provides a completion port for I/O threads (NT platforms only).
- Callback can be bound to files or other system resources.
- Timer and Wait APIs are available.
- The thread pool determines how many threads should be active by using heuristics like delay since last injection, number of current threads, and size of the queue.
- Threads feed from a shared queue.

What's different in .NET:

- It is aware of threads blocking in managed code (e.g. due to garbage collection, managed wait) and can adjust its thread injection logic accordingly.
- There is no guarantee of service for individual threads.

When to Handle Threads Yourself

Using the thread pool effectively is closely linked with knowing what you need from your threads. If you need a guarantee of service, you'll need to manage it yourself. For most cases, using the pool will provide you with the optimal performance. If you have hard restrictions and need tight control of your threads, it probably makes more sense to use native threads anyway, so be wary about handling managed threads yourself. If you do decide to write managed code and handle the threading on your own, make sure that you don't spawn threads on a per-connection basis: this will only hurt performance. As a rule of thumb, you should only choose to handle threads yourself in the managed world in very specific scenarios where there is a large, time-consuming task that is done rarely. One example might be filling a large cache in the background, or writing out a large file to disk.

Tuning for Speed

The thread pool sets a limit on how many threads should be active, and if many of them block then the pool will starve. Ideally, you should use the thread pool for short-lived, non-blocking threads. In server applications, you want to answer each request quickly and efficiently. If you spin up a new thread for every request, you deal with a lot of overhead. The solution is to recycle your threads, taking care to clean up and return the state of every thread upon completion. These are the scenarios were the thread pool is a major performance and design win, and where you should make good use of the technology. The thread pool handles the state cleanup for you, and makes sure the optimal number of threads are in use at a given time. In other situations, it may make more sense to handle threading on your own.

While the CLR can use type-safety to make guarantees about processes to ensure that AppDomains can share the same process, no such guarantee exists with threads. The programmer is responsible for writing well-behaved threads, and all your knowledge from native code still applies.

Below we have an example of a simple application that takes advantage of the thread pool. It creates a bunch of worker threads, and then has them perform a simple task before closing them. I've taken out some error-checking, but this is the same code that can be found in the Framework SDK folder under "Samples\Threading\Threadpool". In this example we have some code that creates a simple work item, and uses the threadpool to have multiple threads handle these items without the programmer having to manage them. Check out the ReadMe.html file for more information.

```
using System;
using System.Threading;

public class SomeState{
  public int Cookie;
  public SomeState(int iCookie){
    Cookie = iCookie;
  }
};


public class Alpha{
  public int [] HashCount;
  public ManualResetEvent eventX;
  public static int iCount = 0;
  public static int iMaxCount = 0;
  public Alpha(int MaxCount) {
    HashCount = new int[30];
    iMaxCount = MaxCount;
  }


   //   The method that will be called when the Work Item is serviced
   //   on the Thread Pool
   public void Beta(Object state){
     Console.WriteLine(" {0} {1} :",
               Thread.CurrentThread.GetHashCode(), ((SomeState)state).Cookie);
     Interlocked.Increment(ref HashCount[Thread.CurrentThread.GetHashCode()]);

     //   Do some busy work
     int iX = 10000;
     while (iX > 0){ iX--;}
     if (Interlocked.Increment(ref iCount) == iMaxCount) {
       Console.WriteLine("Setting EventX ");
       eventX.Set();
     }
  }
};

public class SimplePool{
  public static int Main(String[] args)   {
    Console.WriteLine("Thread Simple Thread Pool Sample");
    int MaxCount = 1000;
    ManualResetEvent eventX = new ManualResetEvent(false);
    Console.WriteLine("Queuing {0} items to Thread Pool", MaxCount);
    Alpha oAlpha = new Alpha(MaxCount);
    oAlpha.eventX = eventX;
    Console.WriteLine("Queue to Thread Pool 0");
    ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),new SomeState(0));
       for (int iItem=1;iItem < MaxCount;iItem++){
         Console.WriteLine("Queue to Thread Pool {0}", iItem);
```

```
        ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),
                                     new SomeState(iItem));
      }
    Console.WriteLine("Waiting for Thread Pool to drain");
    eventX.WaitOne(Timeout.Infinite,true);
    Console.WriteLine("Thread Pool has been drained (Event fired)");
    Console.WriteLine("Load across threads");
    for(int iIndex=0;iIndex<oAlpha.HashCount.Length;iIndex++)
      Console.WriteLine("{0} {1}", iIndex, oAlpha.HashCount[iIndex]);
    }
    return 0;
  }
}
```

The JIT

The Basics

As with any VM, the CLR needs a way to compile the intermediate language down to native code. When you compile a program to run in the CLR, your compiler takes your source from a high-level language down to a combination of MSIL (Microsoft Intermediate Language) and metadata. These are merged into a PE file, which can then be executed on any CLR-capable machine. When you run this executable, the JIT starts compiling the IL down to native code, and executing that code on the real machine. This is done on a per-method basis, so the delay for JITing is only as long as needed for the code you want to run.

The JIT is quite fast, and generates very good code. Some of the optimizations it performs (and some explanations of each) are discussed below. Bear in mind that most of these optimizations have limits imposed to make sure that the JIT does not spend too much time.

- **Constant Folding**—Calculate constant values at compile time.

Before	After
`x = 5 + 7`	`x = 12`

- **Constant- and Copy-Propagation**—Substitute backwards to free variables earlier.

Before	After
`x = a`	`x = a`
`y = x`	`y = a`
`z = 3 + y`	`z = 3 + a`

- **Method Inlining**—Replace args with values passed at call time, and eliminate the call. Many other optimizations can then be performed to cut out dead code. For speed reasons, the current JIT has several boundaries on what it can inline. For example, only small methods are inlined (IL size less than 32), and flow-control analysis is fairly primitive.

Before	After

`?`
`x=foo(4,   true);`
`?`
	`?`
	`x = 9`
	`?`

```
}                          }

foo(int a, bool b){       foo(int a, bool b){
  if(b){                    if(b){
    return a + 5;             return a + 5;
  } else {                  } else {
    return 2a + bar();        return 2a + bar();
}                         }
```

- **Code Hoisting and Dominators**—Remove code from inside loops if it is duplicated outside. The 'before' example below is actually what gets generated at the IL level, since all array indices have to be checked.

  Before **After**

  ```
  for(i=0; i< a.length;i++){  for(int i=0; i<a.length; i++){
    if(i < a.length()){         a[i] = null;
      a[i] = null             }
    } else {
      raise IndexOutOfBounds;
    }
  }
  ```

- **Loop Unrolling**—The overhead of incrementing counters and performing the test can be removed, and the code of the loop can be repeated. For extremely tight loops, this results in a performance win.

  Before **After**

  ```
  for(i=0; i< 3; i++){          print("flaming monkeys!");
    print("flaming monkeys!"); print("flaming monkeys!");
  }                             print("flaming monkeys!");
  ```

- **Common SubExpression Elimination**—If a live variable still contains the information being re-calculated, use it instead.

  Before **After**

  ```
  x = 4 + y x = 4 + y
  z = 4 + y z = x
  ```

- **Enregistration**—It is not useful to give a code example here, so an explanation will have to suffice. This optimization can spend time looking at how locals and temps are used in a function, and try to handle register assignment as efficiently as possible. This can be an extremely expensive optimization, and the current CLR JIT only considers a maximum of 64 local variables for enregistration. Variables that are not considered are placed in the stack frame. This is a classic example of the limitations of JITing: while this is fine 99% of the time, highly unusual functions that have 100+ locals will be optimized better using traditional, time-consuming pre-compilation.
- **Misc**—Other simple optimizations are performed, but the list above is a good sample. The JIT also does passes for dead-code, and other peephole optimizations.

When Does Code Get JITed?

Here is the path your code goes through when it is executed:

1. Your program is loaded, and a function table is initialized with pointers referencing the IL.
2. The **Main** method is JITed into native code, which is then run. Calls to functions get compiled into indirect function calls through the table.
3. When another method is called, the run time looks at the table to see if it points into JITed code.
    a. If it has (perhaps it has been called from another call site, or has been precompiled), the control flow continues.
    b. *If not, the method is JITed and the table is updated.*
4. As they are called, more and more methods are compiled into native code, and more entries in the table point into the growing pool of x86 instructions.
5. As the program runs, the JIT is called less and less often until everything is compiled.
6. A method is not JITed until it is called, and then it is never JITed again during the execution of the program. *You only pay for what you use*.

Myth: JITed Programs Execute Slower than Precompiled Programs

This is rarely the case. The overhead associated with JITing a few methods is minor compared to the time spent reading in a few pages from disk, and methods are JITed only as they are needed. The time spent in the JIT is so minor that it is almost never noticeable, and once a method has been JITed, you never incur the cost for that method again. I'll talk more about this in the Precompiling Code section.

As mentioned above, the version1 (v1) JIT does most of the optimizations that a compiler does, and will only get faster in the next version (vNext), as more advanced optimizations are added. More importantly, the JIT can perform some optimizations that a regular compiler can't, such a CPU-specific optimizations and cache tuning.

JIT-Only Optimizations

Since the JIT is activated at run time, there is a lot of information around that a compiler isn't aware of. This allows it to perform several optimizations that are only available at run time:

- **Processor-Specific Optimizations**— At run time, the JIT knows whether or not it can make use of SSE or 3DNow instructions. Your executable will be compiled specially for P4, Athlon or any future processor families. You deploy once, and the same code will improve along with the JIT and the user's machine.
- **Optimizing away levels of indirection**, since function and object location are available at run time.
- **The JIT can perform optimizations across assemblies**, providing many of the benefits you get when compiling a program with static libraries but maintaining the flexibility and small footprint of using dynamic ones.
- **Aggressively inline functions** that are called more often, since it is aware of control flow during run time. The optimizations can provide a substantial speed boost, and there is a lot of room for additional improvement in vNext.

These run time improvements come at the expense of a small, one-time startup cost, and can more than offset the time spent in the JIT.

Precompiling Code (Using ngen.exe)

For an application vendor, the ability to precompile code during installation is an attractive option. Microsoft does provide this option in the form `ngen.exe`, which will let you run the normal JIT compiler over your whole program once, and save the result. Since the run time-only optimizations cannot be performed during precompilation, the code generated is not usually as good as that generated by a normal JIT. However, without having to JIT methods on the fly, the startup cost is much lower, and some programs will launch noticeably faster. In the future, ngen.exe may do more than simply run the same run time JIT: more aggressive optimizations with higher bounds than the run time, load-order-optimization exposure to developers (optimizing the way code is packed into VM pages), and more complex, time consuming optimizations that can take advantage of the time during precompilation.

Cutting the startup time helps in two cases, and for everything else it doesn't compete with the run time-only optimizations that regular JITing can do. The first situation is where you call an enormous number of methods early on in your program. You'll have to JIT a lot of methods up front, resulting in an unacceptable load time. This is not going to be the case for most people, but pre-JITing might make sense if it affects you. Precompiling also makes sense in the case of large shared libraries, since you pay the cost of loading these much more often. Microsoft precompiles the Frameworks for the CLR, since most applications will use them.

It's easy to use ngen.exe to see if precompiling is the answer for you, so I recommend trying it out. However, most of the time it's actually better to use the normal JIT and take advantage of the run-time optimizations. They have a huge payoff, and will more than offset the one-time startup cost in most situations.

Tuning for Speed

For the programmer, there are really only two things worth noting. First, that the JIT is very intelligent. Don't try to outthink the compiler. Code the way you would normally. For example, suppose you have the following code:

```
?
for(int i = 0; i < myArray.length; i++){
  ?
}

?
```

```
?
int l = myArray.length;
for(int i = 0; i < l; i++){
  ?
}

?
```

Some programmers believe that they can get a speed boost by moving the length calculation out and saving it to a temp, as in the example on the right.

The truth is, optimizations like this haven't been helpful for nearly 10 years: modern compilers are more than capable of performing this optimization for you. In fact, sometimes things like this can actually hurt performance. In the example above, a compiler would probably check to see that the length of **myArray** is constant, and insert a constant in the comparison of the **for** loop. But the code on the right might trick the compiler into thinking that this value must be stored in a register, since `l` is live throughout the loop. The bottom line is: write the code that's the most readable and that makes the most sense. It's not going to help to try to outthink the compiler, and sometimes it can hurt.

The second thing to talk about is tail-calls. At the moment, the C# and Microsoft® Visual Basic® compilers do not provide you with the ability to specify that a tail call should be used. If you really need this feature, one option is to open the PE file in a disassembler and use the MSIL .tail instruction instead. This is not an elegant solution, but tail-calls are not as useful in C# and Visual Basic as they are in languages like Scheme or ML. People writing compilers for languages that really take advantage of tail-calls should be sure to use this instruction. The reality for most people is that even manually tweaking the IL to use tail-calls does not provide an enormous speed benefit. Sometimes the run time will actually change these back into regular calls, for security reasons! Perhaps in future versions more effort will be put in to support tail-calls, but at the moment the performance gain is insufficient to warrant it, and very few programmers will want to take advantage of it.

AppDomains

The Basics

Interprocess communication is becoming more and more common. For stability and security reasons, the OS keeps applications in separate address spaces. A simple example is the manner in which all 16-bit applications are executed in NT: if run in a separate process, one application cannot interfere with the execution of another. The problem here is the cost of the context switch, and opening a connection between processes. This operation is very expensive, and hurts performance a lot. In server applications, which often host several web applications, this is a major drain on both performance and scalability.

The CLR introduces the concept of an AppDomain, which is similar to a process in that it is a self-contained space for an application. However, AppDomains are not restricted to one-per-process. It is possible to run two completely unrelated AppDomains in the same process, thanks to type-safety provided by managed code. The performance boost here is enormous for situations where you normally spend a lot of your execution time in interprocess communication overhead: IPC between assemblies is five times faster than between processes in NT. By reducing this cost dramatically, you get both a speed boost and a new option during program design: it now makes sense to use separate processes where before it may have been far too expensive. The ability to run multiple programs in the same process with the same security as before has tremendous implications for scalability and security.

Support for AppDomains is not present in the OS. AppDomains are handled by a CLR host, such as the ones present in ASP.NET, a shell executable, or Microsoft Internet Explorer. You can also write your own. Each host specifies a *default domain*, which is loaded when the application first launches and is only closed when the process is ended. When you load other assemblies into the process, you can specify that they be loaded into a specific AppDomain, and set different security policies for each of them. This is described in greater detail in the Microsoft .NET Framework SDK documentation.

Tuning for Speed

To use AppDomains effectively, you need to think about what kind of application you're writing, and what kind of work it needs to do. As a good rule of thumb to go by, AppDomains are most effective when your application fits some of following characteristics:

- It spawns a new copy of itself often.
- It works with other applications to process information (database queries inside a web server, for example).
- It spends a lot of time in IPC with programs that work exclusively with your application.
- It opens and closes other programs.

An example of a situation in which AppDomains are helpful can be seen in a complex ASP.NET application. Suppose you want to enforce isolation between different vRoots: in native space, you need to put each vRoot in a separate process. This is fairly expensive, and context switching between them is a lot of overhead. In the managed world, each vRoot can be a separate AppDomain. This preserves the isolation required while drastically cutting the overhead.

AppDomains are something you should use only if your application is complex enough to require working closely with other processes, or other instances of itself. While iter-AppDomain communication is far faster than inter-process communication, the cost of starting and closing an AppDomain can actually be more expensive. AppDomains can end up hurting performance when used for the wrong reasons, so make sure you're using them in the right situations. Note that only managed code can be loaded into an AppDomain, since unmanaged code cannot be guaranteed secure.

Assemblies that are shared between multiple AppDomains must be JITed for every domain, in order to preserve isolation between domains. This results in a lot of duplicate code creation and wasted memory. Consider the case of an application that answers requests with some sort of XML service. If certain requests must be kept isolated from one another, you will need to route them to different AppDomains. The problem here is that every AppDomain will now require the same XML libraries, and the same assembly will be loaded multiple times.

One way around this is to declare an assembly to be *Domain-Neutral*, meaning that no direct references are allowed and isolation is enforced through indirection. This saves time, since the assembly is JITed only once. It also saves memory, since nothing is duplicated. Unfortunately, there is a performance hit due to the indirection required. Declaring an assembly to be domain-neutral results in a performance win when memory is a concern, or when too much time is wasted JITing code. Scenarios like this are common in the case of a large assembly that is shared by several domains.

Security

The Basics

Code access security is a powerful, extremely useful feature. It offers users safe execution of semi-trusted code, protects from malicious software and several kinds of attacks, and allows controlled, identity-based access to resources. In native code, security is extremely difficult to provide, since there is little type-safety and the programmer handles the memory. In the CLR, the run time knows enough about running code to add strong security support, a feature that is new for most programmers.

Security affects both the speed and the working set size of an application. And, as with most areas of programming, how the developer uses security can greatly determine its impact on performance. The security system is designed with performance in mind and should, in most cases, perform well with little or no thought given to it by the application developer. However, there are several things you can do to squeeze ever last bit of performance from the security system.

Tuning for Speed

Performing a security check typically requires a stack walk to make sure that the code calling the current method has the correct permissions. The run time has several optimizations that help it avoid walking the entire stack, but there are several things the programmer can do to help. This brings us to the notion of imperative versus declarative security: declaritive security adorns a type or its members with various permissions, whereas imperative security creates a security object and performs operations on it.

- Declarative security is the fastest way to go for **Assert**, **Deny** and **PermitOnly**. These operations typically require a stack walk to locate the correct call frame, but this can be avoided if you explicitly declare these modifiers. Demands are faster if done imperatively.
- When doing interop with unmanaged code, you can remove the run-time security checks by use the SuppressUnmanagedCodeSecurity attribute. This moves the check into link time, which is much faster. As a note of caution, make sure that the code exposes no security holes to other code, which could exploit the removed check into unsafe code.
- Identity checks are more expensive than code checks. You can use LinkDemand to do these checks at link time instead.

There are two ways to optimize security:

- Perform checks at link time instead of run time.
- Make security checks declarative, rather than imperative.

The first thing you should concentrate on is moving as many of these checks to link time as possible. Bear in mind that this can impact the security of your application, so make sure that you don't move checks into the linker that depend on run-time state. Once you've moved as much as possible into link time, you should optimize the run-time checks by using declarative or imperative security: choose which is optimal for the specific kind of check you use.

Remoting

The Basics

Remoting technology in .NET extends the rich type system and the functionality of the CLR over the network. Using XML, SOAP, and HTTP, you can call procedures and pass objects remotely, just as if they were hosted on the same computer. You can think of this as the .NET version of DCOM or CORBA, in that it provides a superset of their functionality.

This is particularly useful in a server environment, when you have several servers hosting different services, all talking to one another to link those services seamlessly. Scalability is improved too, since processes can be physically broken up across multiple computers without losing functionality.

Tuning for Speed

Since remoting often incurs a penalty in terms of network latency, the same rules apply in the CLR that always have: try to minimize the amount of traffic you send, and avoid having the rest of the program wait for a remote call to return. Here are a few good rules to live by when using remoting to maximize performance:

- **Make Chunky Instead of Chatty Calls**—See if you can cut down the number of calls you have to make remotely. For example, suppose you set some properties for a remote object using **get()** and **set()** methods. It would save you time to simply re-create the object remotely, with those properties set at creation. Since this can be done using a single remote call, you'll save time wasted in network traffic. Sometimes it might make sense to move the object to the local machine, set the properties there and then copy it back. Depending on bandwidth and latency, sometimes one solution will make more sense than the other.
- **Balance CPU Load with Network Load**—Sometimes it makes sense to send something out to be done over the network, and other times it's better to do the work yourself. If you waste a lot of time traversing the network, your performance will suffer. If you use up too much of your CPU, you will be unable to answer other requests. Finding a good balance between these two is essential for getting your application to scale.
- **Use Asynchronous Calls**—When you make a call across the network, make sure it is asynchronous unless you really need otherwise. Otherwise your application will stall until it receives a response, and that can be unacceptable in a user interface or high-volume server. An good example to look at is available in the Framework SDK that ships with .NET, under "Samples\technologies\remoting\advanced\asyncdelegate."
- **Use Objects Optimally**—You can specify that a new object is created for every request (SingleCall) or that the same object is used for all requests (Singleton). Having a single object for all requests is certainly less resource-intensive, but you will need to be careful about the object's synchronization and configuration from request to request.
- **Make Use of Pluggable Channels and Formatters**—A powerful feature of remoting is the ability to plug any channel or formatter into your application. For example, unless you need to get through a firewall, there's no reason to use the HTTP channel. Plugging in a TCP channel will get you much better performance. Make sure you choose the channel or formatter that's best for you.

ValueTypes

The Basics

The flexibility afforded by objects comes at a small performance price. Heap-managed objects take more time to allocate, access, and update than stack-managed ones. This is why, for example, a struct in C++ much more efficient than an object. Of course, objects can do things that structs can't, and are far more versatile.

But sometimes you don't need all that flexibility. Sometimes you *want* something as simple as a struct, and you don't want to pay the performance cost. The CLR provides you with the ability to specify what is called a *ValueType*, and at compile time this is treated just like a struct. ValueTypes are managed by the stack, and provide you with all the speed of a struct. As expected, they also come with the limited flexibility of structs (there is no inheritance, for example). But for the instances where all you need is a struct, ValueTypes provide an incredible speed boost. More detailed information about ValueTypes, and the rest of the CLR type system, is available on the MSDN Library.

Tuning for Speed

ValueTypes are useful only in cases where you use them as structs. If you need to treat a ValueType as an object, the run time will handle boxing and unboxing the object for you. However, this is even more expensive than creating it as an object in the first place!

Here is an example of a simple test that compares the time it takes to create a large number of objects and ValueTypes:

```
using System;
using System.Collections;

namespace ConsoleApplication{
  public struct foo{
    public foo(double arg){ this.y = arg; }
    public double y;
  }
  public class bar{
    public bar(double arg){ this.y = arg; }
    public double y;
  }
class Class1{
  static void Main(string[] args){
    Console.WriteLine("starting struct loop....");
    int t1 = Environment.TickCount;
    for (int i = 0; i < 25000000; i++) {
      foo test1 = new foo(3.14);
      foo test2 = new foo(3.15);
       if (test1.y == test2.y) break; // prevent code from being
       eliminated JIT
    }
    int t2 = Environment.TickCount;
    Console.WriteLine("struct loop: (" + (t2-t1) + "). starting object
       loop....");
    t1 = Environment.TickCount;
    for (int i = 0; i < 25000000; i++) {
      bar test1 = new bar(3.14);
      bar test2 = new bar(3.15);
      if (test1.y == test2.y) break; // prevent code from being
      eliminated JIT
    }
    t2 = Environment.TickCount;
    Console.WriteLine("object loop: (" + (t2-t1) + ")");
    }
```

Try it yourself. The time gap is on the order of several seconds. Now let's modify the program so the run time has to box and unbox our struct. Notice that the speed benefits of using a ValueType have completely disappeared! The moral here is that ValueTypes are used only in extremely rare situations, when you don't use them as objects. It's important to look out for these situations, since the performance win is often extremely large when you use them right.

```
using System;
using System.Collections;

namespace ConsoleApplication{
  public struct foo{
    public foo(double arg){ this.y = arg; }
    public double y;
  }
  public class bar{
    public bar(double arg){ this.y = arg; }
    public double y;
  }
  class Class1{
    static void Main(string[] args){
      Hashtable boxed_table = new Hashtable(2);
      Hashtable object_table = new Hashtable(2);
      System.Console.WriteLine("starting struct loop...");
      for(int i = 0; i < 10000000; i++){
        boxed_table.Add(1, new foo(3.14));
        boxed_table.Add(2, new foo(3.15));
        boxed_table.Remove(1);
      }
      System.Console.WriteLine("struct loop complete.
                                starting object loop...");
      for(int i = 0; i < 10000000; i++){
        object_table.Add(1, new bar(3.14));
        object_table.Add(2, new bar(3.15));
        object_table.Remove(1);
      }
      System.Console.WriteLine("All done");
    }
  }
}
```

Microsoft makes use of ValueTypes in a big way: all the primitives in the Frameworks are ValueTypes. My recommendation is that you use ValueTypes whenever you feel yourself itching for a struct. As long as you don't box/unbox, they can provide an enormous speed boost.

One extremely important thing to note is that ValueTypes require no marshalling in interop scenarios. Since marshalling is one of the biggest performance hits when interoperating with native code, using ValueTypes as arguments to native functions is perhaps the single biggest performance tweak you can do.

Additional Resources

Related topics on performance in the .NET Framework include:

- Tips and Tricks for Performance in .NET Applications

Watch for future articles currently under development, including an overview of design, architectural and coding philosophies, a walkthrough of performance analysis tools in the managed world, and a performance comparison of .NET to other enterprise applications available today.

Appendix: Hosting the Server Run Time

```
#include "mscoree.h"
#include "stdio.h"
#import "mscorlib.tlb" named_guids no_namespace raw_interfaces_only
no_implementation exclude("IID_IObjectHandle", "IObjectHandle")

long main(){
  long retval = 0;
  LPWSTR pszFlavor = L"svr";

  // Bind to the Run time.
  ICorRun timeHost *pHost = NULL;
  HRESULT hr = CorBindToRun timeEx(NULL,
             pszFlavor,
             NULL,
             CLSID_CorRun timeHost,
             IID_ICorRun timeHost,
             (void **)&pHost);

  if (SUCCEEDED(hr)){
    printf("Got ICorRun timeHost\n");

    // Start the Run time (this also creates a default AppDomain)
    hr = pHost->Start();
    if(SUCCEEDED(hr)){
      printf("Started\n");

      // Get the Default AppDomain created when we called Start
      IUnknown *pUnk = NULL;
      hr = pHost->GetDefaultDomain(&pUnk);

      if(SUCCEEDED(hr)){
        printf("Got IUnknown\n");

        // Ask for the _AppDomain Interface
        _AppDomain *pDomain = NULL;
        hr = pUnk->QueryInterface(IID__AppDomain, (void**)&pDomain);

        if(SUCCEEDED(hr)){
          printf("Got _AppDomain\n");

          // Execute Assembly's entry point on this thread
          BSTR pszAssemblyName = SysAllocString(L"Managed.exe");
          hr = pDomain->ExecuteAssembly_2(pszAssemblyName, &retval);
          SysFreeString(pszAssemblyName);

          if (SUCCEEDED(hr)){
            printf("Execution completed\n");

            //Execution completed Successfully
            pDomain->Release();
            pUnk->Release();
            pHost->Stop();
            ->Release();
            return retval;
          }
```

```
        }
        pDomain->Release();
        pUnk->Release();
      }
    }
    pHost->Release();
  }
  printf("Failure, HRESULT: %x\n", hr);

  // If we got here, there was an error, return the HRESULT
  return hr;
}
```

If you have questions or comments about this article, contact Mahesh Prakriya, program manager for .NET Framework performance issues.

© 2001 Microsoft Corporation. All rights reserved. Terms of Use Privacy Statement Accessibility