Essential Performance Facts and .NET Framework Tips

This article provides performance tips gathered from rewriting the C# and Visual Basic compilers in managed code and includes several real examples from the C# compiler. The .NET platform is highly productive for building apps. Powerful and safe languages and a rich collection of libraries make app building highly fruitful. However, with great productivity comes responsibility. You should use all the power of the .NET Framework, but be prepared to tune your code if you're processing a large amount of data such as files or databases.

Why the New Compiler Performance Applies to Your App

Microsoft rewrote the C# and Visual Basic compilers in managed code to provide new APIs for modeling and analyzing code, building tools, enabling much richer, code-aware experiences in Visual Studio. Rewriting the compilers and building Visual Studio experiences on the new compilers revealed very useful performance insights that are applicable to any large .NET app or an app that processes a lot of data. You do not need to know about compilers to understand the insights shown through the examples from the C# compiler.

Visual Studio uses the compiler APIs to build all the great Intellisense features that users love – colorization of identifiers and keywords, syntax completion lists, squiggles for errors, parameter tips, code issues and actions, and so on. Visual Studio provides all this help while end users are typing and changing the code modeled by the compiler.

When your end users interact with an app, they expect the software to be responsive. Typing or command handling should never be blocked. Help should pop up quickly or give up if the user continues typing. An app today should avoid blocking the UI thread with long computations that make the app feel sluggish.

If you want to know more about the new compilers, please visit the open source project.

Just the Facts

Consider these facts when tuning performance and creating responsive .NET Framework apps.

Fact 1: Don’t Prematurely Optimize

Writing code that is more complex than it needs to be incurs maintenance, debugging, and polishing costs. As an experienced programmer, you have an intuitive grasp of how to code a solution to a problem and write more efficient code naturally. However, you may sometimes prematurely optimize your code. For example, you may use a hash table when a simple array would suffice or use complicated caching that may leak memory instead of simply re-computing a value. You should test for performance issues and analyze your code when you find issues.

Fact 2: If You're Not Measuring, You're Guessing

Profiles and measurements don't lie. Profiles show you whether the CPU is fully loaded or if you're blocked on disk I/O. Profiles tell you what kind and how much memory you are allocating and whether your CPU is spending a lot of time in garbage collection (GC).

You should set performance goals for key customer experiences or scenarios in your app and write tests to measure performance. Investigate failing tests by applying the scientific method: use profiles to guide you, hypothesize what the issue might be, and create an experiment or code change that you can test to confirm the hypothesis or reject it. If you establish baseline performance measurements over time with regular testing, you can isolate changes that cause regressions in performance. Approaching performance work in a rigorous way avoids wasting time with code updates you don't need.

Fact 3: Good Tools Make All the Difference

Good tools let you drill quickly into the biggest performance issues (CPU, memory, or disk) and help you locate the code that causes those bottlenecks. Microsoft ships a variety of performance tools such as Visual Studio Profiler, Windows Phone Analysis Tool, and PerfView.

Perfview is free and amazingly powerful for focusing on deeper issues (disk I/O, GC events, memory) such as the issues demonstrated in the examples later. You can capture performance-related Event Tracing for Windows (ETW) events and quickly see per application, per process, per stack, and per thread information. PerfView shows you how much and what kind of memory your app allocates as well as how much which functions or call stacks contribute to the memory allocations. For details, see the very rich help, demos, and linked videos that download with the tool (such as these on channel 9).

Fact 4: It's All about Allocations

You might think that building a responsive .NET Framework application is all about algorithms such as using quick sort instead of bubble sort, but that's not the case. The biggest factor in building a responsive app is allocating memory, especially when your app is very large or processes large amounts of data.

Almost all of the work to build responsive IDE experiences with the new compiler APIs involved avoiding allocations and managing caching strategies. PerfView traces show that the performance of the new C# and Visual Basic compilers is rarely directly related to being CPU bound. The compilers can be I/O bound when reading hundreds of thousands or millions of lines of code, reading metadata, or emitting generated code. The UI thread delays are nearly all due to garbage collector. The .NET Framework GC is highly tuned for performance and does much of its work concurrently while application code executes. However, a single allocation can trigger an expensive collection and stop all threads (see generation 2 collections).

Common Allocations and Examples

The examples in this section have hidden allocations that appear small. However, if a large application executes the example expressions enough times, these expressions can causes hundreds of megabytes,

even gigabytes, of allocations. For example, one-minute tests that simulated a developer's typing in the editor allocated gigabytes of memory before the performance team focused on typing scenarios.

Boxing

Boxing occurs when value types or immediate values that normally live on the stack or in data structures need to be wrapped in an object (that is, allocate an object to hold the data, then return a pointer to the object). The .NET Framework sometimes boxes values due to the signature of a method or the type of a storage location. Wrapping a value type in an object causes memory allocation. The languages and the .NET Framework avoid boxing when possible, but sometimes it happens when you don't expect it. Many boxing operations can contribute megabytes or gigabytes of allocations to your program, which means that you'll garbage collect more often, and it will take longer.

To see boxing in PerfView, open a trace and look at GC Heap Alloc Stacks under your application name (remember, PerfView reports on all processes). If you see types like System.Int32 and System.Char under allocations, you are boxing value types. Choosing one of these types will show the stacks and functions in which they are boxed.

Example 1: string Methods and Value Type Arguments

This sample code illustrates potentially unnecessary and excessive boxing in large systems:

```
public class Logger
{
    public static void WriteLine(string s) { /*...*/ }
}

public class BoxingExample
{
    public void Log(int id, int size)
    {
        var s = string.Format("{0}:{1}", id, size);
        Logger.WriteLine(s);
    }
}
```

This is a logging facility, so an app may call the **Log** function frequently, maybe millions of times. The problem is that the call to **string.Format** resolves to the overload that takes a string and two objects:

```
string.Format(String, Object, Object);
```

This overload requires the .NET Framework to box the **int** values into objects to pass them to this method call. The fix is to call **id.ToString()** and **size.ToString()** and pass all strings (which are objects) to the **string.Format** call. Calling **ToString()** does allocate a string here, but the string allocation is going to happen anyway inside **string.Format**.

You may consider that this basic call to **string.Format** is just string concatenation, so you might write this code instead:

```
var s = id.ToString() + ':' + size.ToString();
```

However, this line of code introduces a boxing allocation because it compiles to:

```
string.Concat(Object, Object, Object);
```

The .NET Framework must box the character literal to invoke **Concat**.

Fix Example 1

The complete fix is simple. Just replace the character literal with a string literal which incurs no boxing because strings are already objects:

```
var s = id.ToString() + ":" + size.ToString();
```

Example 2: enum Boxing

This example was responsible for a huge amount of allocation in the new C# and Visual Basic compilers due to frequent use of enumeration types, especially in dictionaries' lookup operations.

```
public enum Color
{
    Red, Green, Blue
}

public class BoxingExample
{
    private string name;
    private Color color;
    public override int GetHashCode()
    {
        return name.GetHashCode() ^ color.GetHashCode();
    }
}
```

This problem is very subtle. PerfView would report this as **enum.GetHashCode()** boxing because for implementation reasons, this method boxes the underlying representation of the enumeration type. If you look closely in PerfView, you may see two boxing allocations for each call to **GetHashCode.** The compiler inserts one, and the .NET Framework inserts the other.

Fix Example 2

You can easily avoid both allocations by casting to the underlying representation before calling **GetHashCode**:

```
((int)color).GetHashCode()
```

Another common source of boxing on enumeration types is **enum.HasFlag**. The argument passed to **HasFlag** has to be boxed. In most cases, replacing calls to **HasFlag** with a bitwise test is simpler and allocation-free.

Keep the first performance fact in mind (that is, don't prematurely optimize) and don't start rewriting all your code in this fashion. Be aware of these boxing costs, but only change your code after profiling your app and spotting hot spots.

Strings

String manipulations are some of the biggest culprits for allocations, often showing up in PerfView in the top five allocations. Programs use strings for serialization, JSON, and REST. Strings can be programmatic constants for interoperating with systems when you cannot use enumeration types. When your profiling shows that strings are highly impacting performance, be on the lookout for calls to **Format()**, **Concat()**, **Split()**, **Join()**, **Substring()**, and so on. Using **StringBuilder** to avoid the cost of creating one string from many pieces helps, but even allocating the **StringBuilder** object might become a bottleneck that you need to manage.

Example 3: string Operations

The C# compiler had this code that writes the text of a formatted xml doc comment.

```
public void WriteFormattedDocComment(string text)
{
    string[] lines = text.Split(new[] { "\r\n", "\r", "\n" },
                                StringSplitOptions.None);
    int numLines = lines.Length;
    bool skipSpace = true;
    if (lines[0].TrimStart().StartsWith("///"))
    {
        for (int i = 0; i < numLines; i++)
        {
            string trimmed = lines[i].TrimStart();
            if (trimmed.Length < 4 || !char.IsWhiteSpace(trimmed[3]))
            {
                skipSpace = false;
                break;
            }
        }
        int substringStart = skipSpace ? 4 : 3;
        for (int i = 0; i < numLines; i++)
            WriteLine(lines[i].TrimStart().Substring(substringStart));
    }
    else { /* ... */ }
```

You can see that this code does a lot of string manipulation. The code uses productive library methods to split lines into separate strings, trim white space, check whether the argument **text** is an XML documentation comment, and take sub strings from lines.

On the first line inside **WriteFormattedDocComment** the call to **Split()** allocates a new three-element array as the argument every time it's called. The compiler has to emit code to allocate this array each

time. That's because the compiler doesn't know if **Split()** stores the array somewhere where it might be modified by other code, which would affect later calls to **WriteFormattedDocComment**. The call to **Split()** also allocates a string for every line in **text** and allocates other memory to perform the operation.

WriteFormattedDocComment has three calls to **TrimStart()**, two in inner loops that duplicate work and allocations. To make matters worse, the **TrimStart()** overload chosen for no arguments has this signature:

```
namespace System
{
    public class String
    {
        public string TrimStart(params char[] trimChars);
```

This signature means that each call to **TrimStart()** allocates an empty array in addition to the string result.

Lastly, there is a call to **Substring()**, which usually allocates a new string.

Fix Example 3

Unlike the earlier examples, small edits cannot fix these allocations. You need to step back, look at the problem, and approach it differently. For example, you can recognize that the argument to **WriteFormattedDocComment()** is a string that has all the information that the method needs, so the code could do more indexing instead of allocating many partial strings.

The fix is not complete, but you can see how to apply similar fixes for a complete solution. The C# compiler tackled all these allocations with code like this:

```
private int IndexOfFirstNonWhiteSpaceChar(string text, int start) {
    while (start < text.Length && char.IsWhiteSpace(text[start])) start++;
    return start;
}

private bool TrimmedStringStartsWith(string text, int start, string prefix) {
    start = IndexOfFirstNonWhiteSpaceChar(text, start);
    int len = text.Length - start;
    if (len < prefix.Length) return false;
    for (int i = 0; i < len; i++)
    {
        if (prefix[i] != text[start + i]) return false;
    }
    return true;
}

// etc...
```

The first version of **WriteFormattedDocComment()** allocated an array, several sub strings, and a trimmed sub string along with an empty **params** array. It also checked for "///". The revised code uses

only indexing and allocates nothing. It finds the first character that is not white space character and then compares character by character to see if the string starts with "///". Instead of using **TrimStart()**, the new code uses **IndexOfFirstNonWhiteSpaceChar** to return the first index (after a specified start index) where a non-whitespace character occurs. By applying this new approach throughout the code, you can remove all allocations in **WriteFormattedDocComment()**.

Example 4: StringBuilder

This example uses **StringBuilder**. The following function generates a full type name for generic types:

```
public class Example
{
    // Constructs a name like "SomeType<T1, T2, T3>"
    public string GenerateFullTypeName(string name, int arity)
    {
        StringBuilder sb = new StringBuilder();

        sb.Append(name);
        if (arity != 0)
        {
            sb.Append("<");
            for (int i = 1; i < arity; i++)
            {
                sb.Append("T"); sb.Append(i.ToString()); sb.Append(", ");
            }
            sb.Append("T"); sb.Append(i.ToString()); sb.Append(">");
        }

        return sb.ToString();
    }
}
```

The focus is on the line that creates a new **StringBuilder** instance. The code causes an allocation for **sb.ToString()** and internal allocations within the **StringBuilder** implementation, but you cannot control those allocations if you want the string result.

Fix Example 4

The fix for the **StringBuilder** object allocation is to cache it. Even caching a single instance that might get thrown away can improve the performance significantly. This is the function's new implementation, omitting all the code except the new first and last lines:

```
// Constructs a name like "Foo<T1, T2, T3>"
public string GenerateFullTypeName(string name, int arity)
{
    StringBuilder sb = AcquireBuilder();
    /* Use sb as before */
    return GetStringAndReleaseBuilder(sb);
}
```

The key parts are the new **AcquireBuilder()** and **GetStringAndReleaseBuilder()** functions:

```
[ThreadStatic]
private static StringBuilder cachedStringBuilder;

private static StringBuilder AcquireBuilder()
{
    StringBuilder result = cachedStringBuilder;
    if (result == null)
    {
        return new StringBuilder();
    }
    result.Clear();
    cachedStringBuilder = null;
    return result;
}

private static string GetStringAndReleaseBuilder(StringBuilder sb)
{
    string result = sb.ToString();
    cachedStringBuilder = sb;
    return result;
}
```

These implementations use a thread-static field to cache the **StringBuilder** due to the new compilers' use of threading, and you likely can forgo the **ThreadStatic** declaration. The thread-static field holds a unique value for each thread that executes this code.

AcquireBuilder() returns the cached StringBuilder instance if there is one, after clearing it and setting the field or cache to null. Otherwise **AcquireBuilder()** creates a new instance and returns it, leaving the field or cache set to null.

When you're done with the **StringBuilder**, you call **GetStringAndReleaseBuilder()** to get the string result, save the **StringBuilder** in the field or cache, and then return the result. It is possible to re-enter this code and to create multiple **StringBuilder** objects (although that rarely happens), and the code saves only the last released **StringBuilder** instance for later use. This simple caching strategy significantly reduced allocations in the new compilers. Parts of the .NET Framework and MSBuild use a similar technique to improve performance.

This simple caching strategy adheres to good cache design because it has a size cap. There is more code now than in the original, which means more maintenance costs. You should adopt this strategy only after you've found a performance problem, and PerfView has shown you that **StringBuilder** allocations are a relatively big contribution.

LINQ and Lambdas

Using LINQ and lambdas is a great example of using productive features that might require rewriting code if the code executes a large number of times.

Example 5: Lambdas, List<T>, and IEnumerable<T>

This example uses LINQ and functional style code to find a symbol in the compiler's model given a name string:

```
class Symbol {
    public string Name { get; private set; }
    /*...*/
}

class Compiler {
    private List<Symbol> symbols;
    public Symbol FindMatchingSymbol(string name)
    {
        return symbols.FirstOrDefault(s => s.Name == name);
    }
}
```

The new compiler and IDE experiences built on it call **FindMatchingSymbol()** very frequently, and there are several hidden allocations in this function's single line of code. To examine those allocations, first split the function's single line of code into two lines:

```
Func<Symbol, bool> predicate = s => s.Name == name;
return symbols.FirstOrDefault(predicate);
```

In the first line, the lamda expression **"s => s.Name == name"** closes over the local variable **name**. This means that in addition to allocating an object for the delegate that **predicate** holds, a static class is allocated to hold the environment that captures the value of **name**. The compiler generates code like the following:

```
// Compiler-generated class to hold environment state for lambda
private class Lambda1Environment
{
    public string capturedName;
    public bool Evaluate(Symbol s)
    {
        return s.Name == this.capturedName;
    }
}

// Expanded Func<Symbol, bool> predicate = s => s.Name == name;
Lambda1Environment l = new Lambda1Environment() { capturedName = name };
var predicate = new Func<Symbol, bool>(l.Evaluate);
```

The two **new** allocations (one for the environment class and one for the delegate) are explicit now.

Now look at the call to **FirstOrDefault**, which is an extension method on **IEnumerable<T>** and incurs an allocation too. Because **FirstOrDefault** takes an **IEnumerable<T>** object as its first argument, you can expand the call to the following code (simplified a bit for discussion):

```
// Expanded return symbols.FirstOrDefault(predicate) ...
IEnumerable<Symbol> enumerable = symbols;
```

```
IEnumerator<Symbol> enumerator = enumerable.GetEnumerator();
while(enumerator.MoveNext())
{
    if (predicate(enumerator.Current))
        return enumerator.Current;
}
return default(Symbol);
```

The **symbols** variable has type List<T>. The **List<T>** collection type implements **IEnumerable<T>** and cleverly defines an enumerator that **List<T>** implements with a struct. Using a struct instead of a class means that you usually avoid any heap allocations, which, in turn, can affect garbage collection performance. Enumerators are typically used with the language's **foreach** loop, which uses the enumerator struct as it is returned on the call stack. Incrementing the call stack pointer to make room for an object does not affect GC the way a heap allocation does.

In the case of the expanded **FirstOrDefault** call, the code needs to call **GetEnumerator()** on an **IEnumerable<T>**. Assigning **symbols** to the **enumerable** variable of type **IEnumerable<Symbol>** loses the information that the actual object is a **List<T>**. This means that when the code fetches the enumerator with **enumerable.GetEnumerator()**, the .NET Framework has to box the returned struct to assign it to the **enumerator** variable.

Fix Example 5

The fix is to rewrite FindMatchingSymbol as follows, replacing its single line of code with six lines of code that are still succinct, easy to read and understand, and easy to maintain:

```
public Symbol FindMatchingSymbol(string name)
    {
        foreach (Symbol s in symbols)
        {
            if (s.Name == name)
                return s;
        }
        return null;
    }
```

This code doesn't use LINQ extension methods, lambdas, or enumerators, and it incurs no allocations. There are no allocations because the compiler can see that the **symbols** collection is a **List<T>** and can bind the resulting struct enumerator to a local variable with the right type to avoid boxing the struct. The original version of this function was a great example of the expressive power of C# and the productivity of the .NET Framework. This new and more efficient version is also simple and does not add complex code to maintain.

Async

The next example shows a common problem when you try to use cached results in an async method.

Example 6: Caching in Async Methods

The Visual Studio IDE features built on the new C# and Visual Basic compilers fetch syntax trees a lot, and the compilers use **async** when doing so to keep Visual Studio responsive. Here's the first version of the code you might write to get a syntax tree:

```
class SyntaxTree { /*...*/ }

class Parser { /*...*/
    public SyntaxTree Syntax { get; }
    public Task ParseSourceCode() { /*...*/ }
}

class Compilation { /*...*/
    public async Task<SyntaxTree> GetSyntaxTreeAsync()
    {
        var parser = new Parser(); // allocation
        await parser.ParseSourceCode(); // expensive
        return parser.Syntax;
    }
}
```

You can see that calling **GetSyntaxTreeAsync()** instantiates a **Parser**, parses the code, and then returns a **Task<SyntaxTree>**. The expensive part is allocating the **Parser** instance and parsing the code. The function returns a **Task** so that callers can await the parsing work and free the UI thread to be responsive to user input.

Because several Visual Studio features might try to get the same syntax tree, you might cache the parsing result to save time and allocations. However, the following code that you would likely write incurs an allocation:

```
class Compilation { /*...*/

    private SyntaxTree cachedResult;

    public async Task<SyntaxTree> GetSyntaxTreeAsync()
    {
        if (this.cachedResult == null)
        {
            var parser = new Parser(); // allocation
            await parser.ParseSourceCode(); // expensive
            this.cachedResult = parser.Syntax;
        }
        return this.cachedResult;
    }
}
```

You see that the new code with caching has a **SyntaxTree** field named **cachedResult**. When this field is null, **GetSyntaxTreeAsync()** does the work and saves the result in the cache. **GetSyntaxTreeAsync()** returns the **SyntaxTree** object. The problem is that when you have an **async** function of type **Task<SyntaxTree>**, and you return a value of type **SyntaxTree**, the compiler emits code to allocate a **Task** to hold the result (by using **Task<SyntaxTree>.FromResult()**). The **Task** is marked as completed, and the result is immediately available. Allocating **Task** objects that were already completed occurred so often that fixing this allocation noticeably improved responsiveness.

Fix Example 6

To remove the completed Task allocation, you can cache the Task object with the completed result:

```
class Compilation { /*...*/

    private Task<SyntaxTree> cachedResult;

    public Task<SyntaxTree> GetSyntaxTreeAsync()
    {
        return this.cachedResult ??
               (this.cachedResult = GetSyntaxTreeUncachedAsync());
    }

    private async Task<SyntaxTree> GetSyntaxTreeUncachedAsync()
    {
        var parser = new Parser(); // allocation
        await parser.ParseSourceCode(); // expensive
        return parser.Syntax;
    }
}
```

This code changes the type of **cachedResult** to **Task<SyntaxTree>** and employs an **async** helper function that holds the original code from **GetSyntaxTreeAsync()**. **GetSyntaxTreeAsync()** now uses the null coalescing operator to return **cachedResult** if it is not null. If **cachedResult** is null, then **GetSyntaxTreeAsync()** calls **GetSyntaxTreeUncachedAsync()** and caches the result. Notice that **GetSyntaxTreeAsync()** doesn't await the call to **GetSyntaxTreeUncachedAsync()** as the code would normally. Not using await means that when **GetSyntaxTreeUncachedAsync()** returns its **Task** object, **GetSyntaxTreeAsync()** immediately returns the **Task**. Now, the cached result is a **Task**, so there are no allocations to return the cached result.

Miscellaneous Quick Mentions

There are a few more points about potential problems in large apps or apps that process a lot of data.

Dictionaries

Dictionaries are used ubiquitously in many programs, and though dictionaries are very convenient and inherently efficient, they're often used inappropriately. In Visual Studio and the new compilers, analysis shows that many of the dictionaries contained a single element or were empty. An empty **Dictionary** has ten fields and occupies 48 bytes on the heap on an x86 machine. Dictionaries are great when you

need a mapping or associative data structure with constant time lookup. However, when you have only a few elements, you waste a lot of space by using a **Dictionary**. Instead, you could iteratively look through a **List<KeyValuePair<K,V>>** , for example, just as fast. If you use a **Dictionary** only to load it with data and then read from it (a very common pattern), using a sorted array with an N(log(N)) lookup might be nearly as fast, depending on your typical number of elements.

Classes vs. Structs

Classes and structs provide a classic space/time tradeoff, loosely speaking, for tuning your apps. Classes incur 12 bytes of overhead on an x86 machine even if they have no fields, but they are cheap to pass around because it only takes a pointer to refer to a class instance. Structs incur no heap allocations if they aren't boxed, but when you pass large structs as function arguments or return values, it takes CPU time to atomically copy all the data members of the structs. Watch out for repeated calls to properties that return structs, and cache the property's value in a local variable to avoid lots of data copying.

Caches

A common performance trick is to cache results. However, a cache without a size cap or disposal policy can be a memory leak. When processing large amounts of data, if you hold on to a lot of memory in caches, you can cause garbage collection to override the benefits of your cached lookups.

Conclusions

With large systems or systems that process a large amount of data, you should be aware of performance bottleneck symptoms that can affect your app's responsiveness at scale – boxing, string manipulations, LINQ and lambda, caching in async methods, caching without a size limit or disposal policy, inappropriate use of Dictionary, and passing around structs. Keep in mind the four facts for tuning your applications:

- Don't prematurely optimize – be productive and tune when you spot problems.
- Profiles don't lie – you're guessing if you're not measuring.
- Good tools make all the difference – download PerfView and look at the tutorials.
- Allocations are king for app responsiveness – this is where the new compilers' perf team spent most of their time.

See Also

If you prefer to watch a presentation on this subject, see the Channel 9 video.

VS Profiler Basics -- http://msdn.microsoft.com/en-us/library/ms182372.aspx

.NET App Performance Tools Overview -- http://msdn.microsoft.com/en-us/library/hh156536.aspx

Windows Phone Performance Analysis Tool -- http://msdn.microsoft.com/en-us/magazine/hh781024.aspx

VS Profiler Extend Example -- http://msdn.microsoft.com/en-us/magazine/cc337887.aspx

A Few C# and VB Performance Tips -- http://msdn.microsoft.com/en-us/library/ms173196(v=vs.110).aspx

Some High-level Tips -- http://curah.microsoft.com/4604/improving-your-net-apps-startup-performance