

ETD Archive

2012

Productivity at the Cost of Efficiency: an Analysis of Advanced C# Programming

Andrew Darovich
Cleveland State University

Follow this and additional works at: <https://engagedscholarship.csuohio.edu/etdarchive>



Part of the [Computer Sciences Commons](#)

[How does access to this work benefit you? Let us know!](#)

Recommended Citation

Darovich, Andrew, "Productivity at the Cost of Efficiency: an Analysis of Advanced C# Programming" (2012). *ETD Archive*. 361.

<https://engagedscholarship.csuohio.edu/etdarchive/361>

This Thesis is brought to you for free and open access by EngagedScholarship@CSU. It has been accepted for inclusion in ETD Archive by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.

PRODUCTIVITY AT THE COST OF EFFICIENCY: AN ANALYSIS OF
ADVANCED C# PROGRAMMING

ANDREW DAROVICH

Bachelor of Science in Computer and Information Science
Cleveland State University

December 2009

submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN COMPUTER AND INFORMATION SCIENCE

at the

CLEVELAND STATE UNIVERSITY

May 2012

This thesis has been approved
For the department of COMPUTER SCIENCE
and the College of Graduate Studies by

Thesis Chairperson, Dr. Ben Blake

Department and Date

Dr. Timothy Arndt

Department and Date

Dr. Haodong Wang

Department and Date

PRODUCTIVITY AT THE COST OF EFFICIENCY: AN ANALYSIS OF
ADVANCED C# PROGRAMMING

ANDREW DAROVICH

ABSTRACT

In this modern age of computer programming, there are many advanced features at our disposal. These are designed with elegance in mind and are put in place to allow programmers to be more productive. They are often meant to remove the need to know machine and system specifics so that programmers can focus on the higher level tasks at hand.

What this analysis focuses on is examining what happens behind the scenes when using these advanced features. Performance for various new features of C# such as anonymous methods, reflection, and iterators were examined alongside more traditional programming styles in order to determine if these advanced features designed for productivity have any negative impacts on program efficiency.

The outcome of this analysis is that these new features are highly beneficial and should be used whenever possible as they have a negligible effect on efficiency. Even when used haphazardly, these new features have proven to be just as efficient as standard programming methods.

TABLE OF CONTENTS

	PAGE
ABSTRACT.....	iii
LIST OF TABLES.....	vi
LIST OF FIGURES.....	vii
CHAPTER	
I. INTRODUCTION.....	1
1.1 The Rise of C#	1
1.2 Scope of Research.....	4
1.2.1 Related Research.....	6
1.3 Text Overview.....	7
II. ANONYMOUS PROGRAMMING.....	9
2.1 Delegates.....	9
2.1.1 Action and Func.....	15
2.2 Anonymous Methods.....	16
2.2.1 Lambdas.....	18
2.3 Analysis Versus Procedural Programming..	20
2.3.1 Anonymous Programming.....	20
2.3.2 Lambdas.....	29
III. DYNAMIC PROGRAMMING.....	42
3.1 The Common Language Runtime.....	42
3.1.1 Just In Time Compilation	44
3.1.2 Reflection.....	54
3.2 Iterators Via Yield.....	66

IV.	CONCLUDING REMARKS.....	76
	4.1 Final Verdict.....	76
	BIBLIOGRAPHY.....	82
	APPENDICES.....	85
	A. Full Class Listings and Disassemblies.....	86

LIST OF TABLES

Table	Page
I. Test System.....	5
II. Anonymous vs. Std. Procedural.....	22
III. Anonymous vs. Std. Procedural w/ Longer Processes..	27
IV. Lambda vs. Procedural.....	31
V. Lambda LINQ Search vs. Std. Procedural Search.....	36
VI. Lambda LINQ Sort vs. Std. Procedural Sort.....	36
VII. Local Variable vs. Array's Length Property.....	47
VIII. Local Variable vs. List's Count Property.....	50
IX. New Operator Vs. Reflection.....	65
X. List Use Vs. Yield Return.....	68
XI. Memory Consumption For Yield Return.....	69

LIST OF FIGURES

Table	Page
1. Simple Delegate Signature.....	9
2. Delegates In Action With Resultant Output.....	11
3. The Disassembly of a Delegate.....	12
4. A Simple Lambda to Cube a Number.....	18
5. Various Lambdas in Action.....	19
6. Lambdas Within Extension Methods.....	19
7. Performing Calculations w/ Anonymous and Standard Procedural Programming.....	21
8. Abridged MSIL Disassembly.....	25
9. Lambda and Procedural Searches.....	30
10. Inefficient Lambdas in Procedural Form.....	32
11. Searching and Sorting with Lambdas and Procedural Programming.....	35
12. MSIL Disassembly of Searching & Sorting w/ Lambdas & Procedural Programming.....	37
13. Reflecting Upon LINQ Extension Methods.....	39
14. List.Sort()'s Behind the Scenes Footage.....	40
15. Local/Non Local Lengths.....	47
16. Local Variable vs. Count Property.....	50
17. Using Reflection to Access Private Methods.....	55
18. Retrieving Handlers Without Reflection.....	60
19. Dynamically Generating Handlers Via Reflection.....	61

20. Effective Use of Yield Return.....	67
21. The State Machine for Yield Return.....	71

CHAPTER I

INTRODUCTION

1.1 The Rise of C#

In the world of computer programming, C quickly emerged as the language of choice for everything from operating systems to video games. This language gave the programmer the power to construct sophisticated programs without having to interact directly with the CPU. One of the finest examples of the power of C is the UNIX operating system. Another example is the groundbreaking, revolutionary Doom engine, written by John Carmack of id Software.

This flexibility and power was not without problems. The programmer was left to manage his or her own memory use. It was also up to the programmer to create his or her own library to perform various algorithms.

The solution to these problems arrived in the form of C++. As the name implies, C++ is simply "C plus 1". With C++, the programmer could make use of the new *Standard Template Library* to perform many algorithms and operations with ease. The programmer was also given some facilities to provide cleaner memory management. Most notably, C++ introduced classes to the realm of programming. Now, a programmer could construct truly object oriented programs.

C++ reigned as king for over a decade. Its versatility has caused it to remain heavily in use today in many different fields. However, because of its C-based roots, it still falls prey to memory management issues, among other problems; the biggest of which is portability. In the current age of programming, portability is a highly desirable trait.

Of the various portable languages, C# has emerged as quite a powerhouse, standing toe to toe with Java. As of May 2012, C# is the 5th most popular language on the TIOBE Index, bested only by C, Java, C++, and Objective C. This is no surprise considering these languages have been out considerably longer than C# and are more established. However, this does not mean that C# is to be taken lightly. It is strongly tied into the .NET framework. As a result, it has many useful features that allow programs to be

deployed to various platforms without modification to the source code. These range from anonymous programming, to dynamic runtimes.

Much like Java, C# is known as a “managed” language. This means that a programmer can make use of all of the powerful features of the language without ever having to concern himself with the memory and machine specific details of the target platform. Like the Java Virtual Machine used with Java programming, C# makes use of the Common Language Runtime (CLR) which allows for various assemblies to be made from the C# program and deployed to any compatible architecture. No modifications to the source code are required. The code compiles into Intermediate Language (IL), which is then passed into the CLR (or JVM with Java). From here, the IL is then translated to machine language for the target architecture.

This feature and many others are part of what makes C# a highly effective language. However, there may still be problems, even with such a feature filled language.

1.2 Scope of Research

The features of C# that make it managed and highly versatile can also have a negative impact on the programs efficiency. What may result in more productivity for the programmer could also mean less efficiency for the program itself. The fact remains that *something* has to be doing all of the memory management and type casting. The dynamic qualities of the language are magic in the literal sense. That is, the real work (the trick) is hidden behind the scenes and the programmer is only exposed to the clean-cut code that results from it. What this means is that if the programmer is not doing it, leaving it all up to C# and its managed features, the program may take efficiency hits at run time. These efficiency hits could have possibly been avoided by taking care of all of these details beforehand using standard procedural programming methods that have been in use for over 30 years.

To prove this point, the research that will be covered herein will focus on some of the more predominant features of the language in detail. Sample programs using features such as lambdas and reflection will be created and run alongside their procedural programming counterparts. Timings for each will be gathered and compared. Also, the programs will be disassembled so that the resultant

Intermediate Language (IL) can be analyzed. This will allow us to see the “magic” that goes on behind the scenes.

The target outcome is to prove that the features of C# that make it a desirable language do in fact have a negative impact on the overall program efficiency and should only be used when the circumstances truly call for it.

All tests will be run on a machine with the following specifications:

System	
Processor:	AMD Phenom(tm) II X4 965 Processor 3.40 GHz
Installed memory (RAM):	4.00 GB (3.75 GB usable)
System type:	64-bit Operating System

Table I - Test System

It should be noted that standard benchmarking programs need not be utilized for the research herein. This is because we are examining the effects of runtime and memory use caused by the features researched within the scope of this paper. This means we will be examining their runtimes and memory consumption in comparison to the rest of the experimental programs used to gather timings. I/O bound and CPU intensive applications alike will both be affected the same by the runtime incurred from the features examined.

1.2.1 Related Research

There does not seem to be a great deal of direct IL disassembly published to date, or direct timing comparisons. Instead, it seems that the focus is on JIT compilation optimization strategies. Therefore, in order to explore the features in question more thoroughly, research in the fields of various compiler optimizations such as "just in time" (JIT) compilation within languages such as C# and Java will also be examined.

The reason for this is that optimizations performed by the compiler will have an impact on how important it is for the programmers themselves to actually perform these tasks.

It may be the case that in the rapidly evolving computer programming world, the behind the scenes activities that the framework does for the programmer are doing a job of creating efficient programs without sacrificing readability and maintainability that is often destroyed by optimizations carried out by the programmer. These advances may be closing the book on the old way of programming and instead opening a new world of dynamic, flexible, and still efficient programming.

Researching the materials that have focused on these types of machine created optimizations will be a crucial aid in determining if this hypothesis is in fact true.

1.3 Text Overview

Because the majority of the research that will be performed focuses on the inner workings of C#, most of the texts used are of the reference nature. These books are directly from Microsoft, and this should ensure that the most up to date information is used. The Microsoft Developer Network (MSDN) and its various publications will also be referenced frequently, as this is the most accurate source of information regarding C#.

With that in mind, the first step is to analyze the anonymous programming paradigm in detail. This will include delegates and their various shorthand approaches, along with anonymous methods.

We will begin by performing a brief overview of the approaches, while hinting at possible efficiency issues that may arise.

Once we have detailed all of the anonymous programming methods, we will compare them to their procedural equivalents to see which performs faster, and why.

After this is completed, the dynamic programming capabilities of C# will be examined in the same manner. Dynamic programming with respect to this paper means code that is generated dynamically at run-time. This is not to be confused with the dynamic programming concept used to

solve a complex problem by subdividing it into smaller, simpler problems, and combining these smaller solutions to form the whole solution.

Investigation into dynamic programming will include investigating reflection, the CLR, and yield return statements in detail to examine how they may be utilized to generate code dynamically for us.

CHAPTER II

ANONYMOUS PROGRAMMING

2.1 Delegates

Before anonymous programming can be fully explored, one must first understand the concept of delegates within C# as they are the key component to an anonymous method. Delegates are C#'s answer to the function pointers found in C/C++. As we will soon see, they are not the exact equivalent. Per the MSDN, delegates allow methods to be passed as parameters, can be used to define callback methods, and can be chained together.

A delegate (as shown in Figure 1 below) takes the following form:

```
public delegate int aDelegate(int value1, int value2);
```

Figure 1: A simple delegate signature

What this will then allow the programmer to do is assign methods with similar signatures to it to perform operations.

With the above example, what we can do now is create a class named **delegateClass** which is full of various mathematical functions that operate on two integers: **add**(int x, int y), **sub**(int x, int y), **mul**(int x, int y), **div**(int x, int y), and **pow**(int x, int y). This is a very contrived example, but it demonstrates the properties of a delegate quite clearly.

Now, we are able to create an instance of our **delegateClass** class, and pass the methods inside into various delegates, as demonstrated below in Figure 2. The entire **delegateClass** definition appears in the Appendix A for further reference.

```
public int mul(int x, int y)
{
    return x * y;
}

class Program
{
    static void Main(string[] args)
    {
        delegateClass dc = new delegateClass();

        aDelegate add = new aDelegate(dc.add);
        aDelegate sub = new aDelegate(dc.sub);
        aDelegate div = new aDelegate(dc.div);
        aDelegate mul = new aDelegate(dc.mul);
        aDelegate pow = new aDelegate(dc.pow);

        Console.WriteLine("Add: " + add(16, 22));
        Console.WriteLine("Sub: " + sub(45, 23));
        Console.WriteLine("Div: " + div(9, 3));
        Console.WriteLine("Mul: " + mul(19, 88));
        Console.WriteLine("Exp: " + pow(2, 3));
    }
}
```

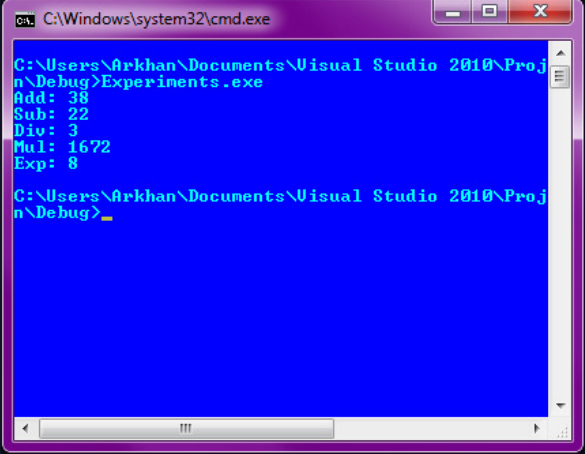


Figure 2: Delegates in action with resultant output

With this simple example demonstrating delegates shown, we can now explore the inner workings of a delegate. As Wagner states in his book, *Effective C#*, a delegate is most commonly used for event driven programming, typically in the form of callbacks.

The reason these are not an exact equivalent to a function pointer also comes from something noted by Wagner that prompted further investigation. Wagner states: "Delegates are objects that reference a method". So, rather than being a simple pointer, they are a class that *contains* a pointer! Disassembling the aforementioned delegate example proves this to be the case as shown in Figure 3. The delegate keyword is converted into a class, which then contains a method called **Invoke ()** which is our

reference method. This convolutes things quite significantly. Thankfully, and most importantly with respect to our research, the work is all done behind the scenes by the compiler.

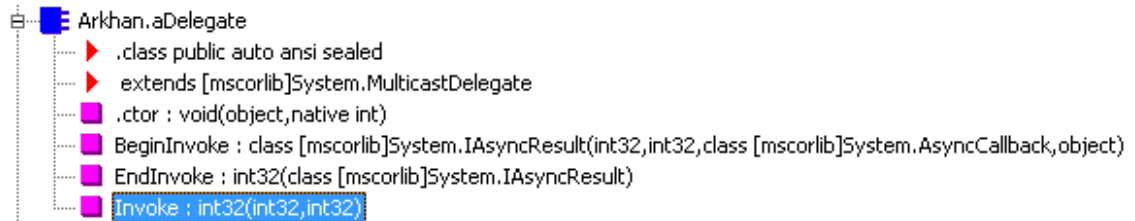


Figure 3: The disassembly of a delegate

This sort of setup is a requirement of C#, being that it is a managed programming language. The programmer does not have the luxury of communicating directly with addresses due to *type safety*. As Jeffrey Richter points out in his article, “An Introduction to Delegates”, delegates are commonly used as a callback mechanism, which agrees with the explanation given by Wagner. The delegate construct carries with it the number and types of parameters expected by the function, the return types, and calling conventions. This provides a programmer with the type safety required by C#; the unsafe possibilities of pointers to functions possible (and probable) within C/C++ are a distant memory in the land of C#.

The interesting design of a delegate also allows for the concept of delegate chaining. An article entitled “Internals of Delegate Chaining” by Aarthi Saravanakumar

details this concept quite well. Inside this article we see that delegates are chained together by the use of a **Combine ()** method. The overall result of delegate chaining is a linked list full of cloned versions of all the methods that have been chained. This detail alone begins to build up concern about the impact of using delegates within a program where efficiency is desired. Use of the **Clone** method results in a shallow copy or deep copy of the object in question. Cloning operates in $O(n)$ time (per the MSDN), which implies that the more you do it, the more time you will spend. This is in contrast to function pointers in C/C++, which do not clone anything, and simply reside in memory, ready to be used when needed.

Richter's article contains a final segment (Demystifying Delegates) that delves deep into the complexities of delegates with the goal being to explain how to use delegates efficiently. In this portion, we again see the now known fact that delegates are not simple pointers. We also again see the **Invoke ()** method. Richter elaborates on this detail to explain what actually happens when a delegate is called. The compiler generates the code to call **Invoke ()** for you since the method in question does not actually exist. Programmers themselves are not allowed to call **Invoke ()** explicitly. Richter also states that the

compiler and the CLR (Common Language Runtime, more on this later.) hide the complexity of delegates on purpose and do the processing for us so that we can focus on the design of our programs rather than the complexities of the system. This confirms that concerns about the existence of behind the scenes work are valid, and should be investigated further.

2.1.1 Action and Func

The one downside to delegates thus far has been with the setup of them. This setup can often negate the supposed elegance of delegates. To remedy this, C# introduces two keywords, **Action** and **Func**, which allow you to forego the usual setup of a delegate and keep it inline.

Actions are a type of delegate that can be used to pass a method as a parameter without ever explicitly declaring a custom delegate. It can be seen as a sort of short hand for delegate declarations. Actions have no return types, and take in no parameters. They are essentially the ultimate solution for quick, parameter-less void functions.

A similar keyword, **Func**, operates the same, yet again acting as shorthand for delegates. However, with **Func**, you are able to specify a return type. Both of these keywords serve to clean up delegate use and make it much more streamlined. However, we must not forget that this shorthand does not mean the work involved with setting up a delegate is gone. It just means we have passed the baton to the compiler yet again.

2.2 Anonymous Methods

We have already seen the core building block of anonymous programming in action, but now it is time to explore the concept further through *Anonymous Methods*, a newer feature to C# that allows the use of delegates without defining named methods.

As the name implies, an anonymous method has no name. It is simply defined in-line and placed right into a delegate. Because of this, the programmer is then leaving a significant amount of the work up to the compiler. This includes inferring the type, and performing the wrapping required of delegates in C#.

An article by Juval Lowy in the MSDN magazine entitled "Create Elegant Code With Anonymous Methods, Iterators, And Partial Classes" details the common uses of anonymous methods. This includes using them in place of delegates in places where a delegate type is the expected input. This article also serves to point out that there are in fact many ways a programmer can create and use anonymous methods.

Unfortunately, as stated by Lowy, the resultant MSIL (Microsoft Intermediate Language) generated by the compiler can be quite different for each different approach to using anonymous methods.

In the case of the anonymous method using class member variables and method arguments, Lowy demonstrates that the code generated is fairly compact. What we will see is the addition of a private method to the class followed by the standard delegate instantiation. It is fairly cut and dry, with minimal overhead.

The issues begin when the anonymous method wishes to use outer variables, meaning local variables or parameters from the containing method. In this case, the compiler does far more work.

First the compiler creates a private nested class with a back reference to the containing class. This nested class contains public member variables corresponding to every single outer variable that is used. Next, the compiler creates a public method with a signature matching the delegate in question. Then, the compiler replaces the anonymous method definition that sparked this entire effort with this nested class. This means it must also take care of all of the assignments for the cloned outer variables. *Finally*, the compiler creates a new delegate object, wraps the public method from the nested class, and calls the delegate, thus invoking the method. With all of the processing required, one can begin to see some of the potential pitfalls of anonymous methods.

2.1.1 Lambdas

Another way a programmer can make use of anonymous functionality is through the use of lambdas. Lambdas may be a new feature to C#. However, they are not a new programming concept. Functional programming languages such as LISP have been making use of lambdas for decades now.

Lambdas in C# provide a very simple, very elegant way to define the anonymous functions we have already covered. They are also a key component to using LINQ extension methods within C#.

In the traditional sense, a lambda takes a form that avoids ambiguity by having you define the number and order of the parameters, as shown in Figure 4.

```
x => x * x * x
```

Figure 4: A Simple Lambda to Cube a Number

The simplicity of a lambda allows you to define complex functions easily so that you may then explore function implementations and computation easily. This also allows for nested expressions, which are part of why lambdas are regarded as very elegant.

Figure 5 demonstrates a few cases of using a lambda in place of the standard anonymous function style. The results are pretty streamlined.

```
String hello = "Hello World!";
Action helloLambda = () => Console.WriteLine(hello); // an Action is a void function

helloLambda(); // calling this will execute the lambda action

// Use Func<parameters,..., returntype> instead of Action if it returns something.
Func<int, int> squarer = i => i * i;
Console.WriteLine(squarer(54)); // Gives us 2916

// Here we have local variables and more than a single
// statement, so we put the declaration in {}'s
Func<int, int> randomizer = i =>
{
    Random rng = new Random();
    int random = rng.Next(i * i);
    return random;
};
Console.WriteLine(randomizer(1622));
```

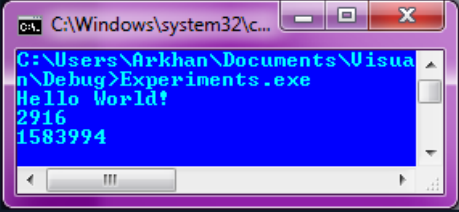


Figure 5: Various lambdas in action.

One of the more predominant uses of lambdas in C# comes from the use of LINQ extension methods. These standard query methods often expect a lambda to be used as the passed in parameter, as shown in figure 6:

```
string title = books.Find(s => s.Contains("Dragon"));
```

Figure 6: Lambdas within extension methods.

This again serves as proof that lambdas can allow for some very clean, very concise code. What the code is doing need not even be explained through comments within the code or within this document. It is alarmingly apparent that this statement searches through books (presumably a list of strings), and tries to find a string containing "Dragon" that it can assign to title. This sort of feature of C# is what makes anonymous programming very desirable.

2.3 Analysis Versus Procedural Programming

2.3.1 Anonymous Programming

Now that we have the basic overview of anonymous programming, complete with possible pitfalls brought to attention, we can begin to dissect these features.

We will be performing various operations with anonymous programming, gathering the timings (in milliseconds), and then comparing them to the same thing done the standard (procedural way).

Our first experiment is using anonymous methods to perform some calculations. We will be using two anonymous styles (one with outer variables used and one without), and one standard procedural style.

The source code appears in Figure 7, and the timings appear in Table II. Assume that **MyDelegate** and **MyDelegate2** are simple void delegates with no parameters.

```

////////////////////////////////////
// Version using outer variable.
MyDelegate d = null;
for (int i = 1; i <= 5; i++)
{
    MyDelegate tempD = delegate
    {
        doABunchOfStuff(i);
    };
    d = tempD;           // set our delegate to call our anonymous method...
    d();                 // call it.
}
sw.Stop();
Console.WriteLine("Elapsed Outer Variable Time : {0}", sw.ElapsedTicks);

sw.Reset();
sw.Start();
////////////////////////////////////
// Version using a delegate chain.
MyDelegate2 d2 = null;
for (int i = 1; i <= 5; i++)
{
    int k = i;           // due to scoping, we must do this, or
                        //the delegate chain we create will print the end value of i every time...
    MyDelegate2 tempD = delegate
    {
        doABunchOfStuff(k);
    };
    d2 += tempD;        // this time, chain the delegates
}
d2();                   // done chaining. Call it.
sw.Stop();
Console.WriteLine("Elapsed Delegate Chain Time: {0}", sw.ElapsedTicks);

sw.Reset();
sw.Start();
////////////////////////////////////
// Standard procedural version.
for (int i = 1; i <= 5; i++)
{
    doABunchOfStuff(i);
}
sw.Stop();
Console.WriteLine("Elapsed Procedural Time: {0}", sw.ElapsedTicks);

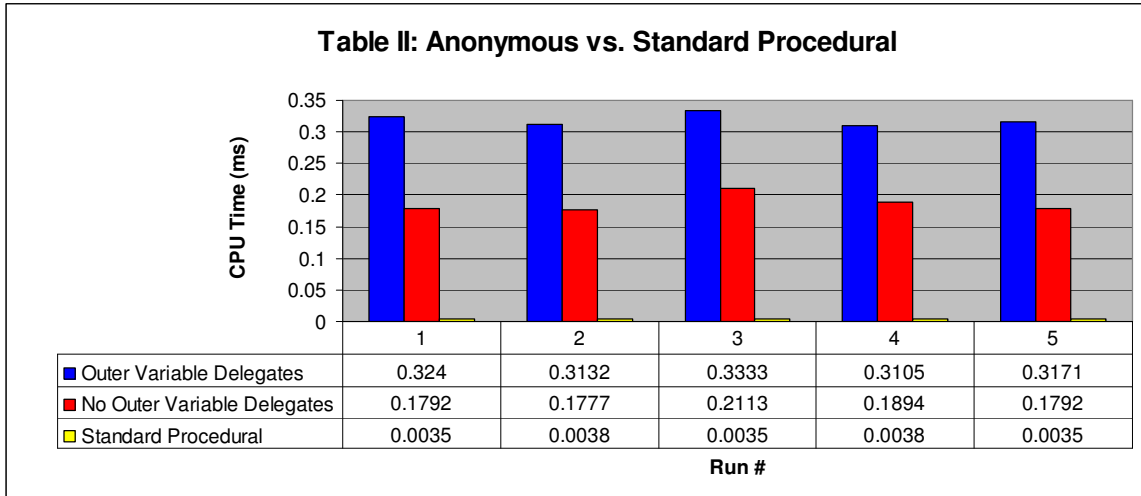
public static void doABunchOfStuff(int i)
{
    double total;

    total = Math.Pow((double) i, (double) i);
    total *= Math.Pow(total, i);

    for (int j = 0; j < 40; j++)
    {
        total += (j + i) << 2;
        total *= total;
    }
    Thread.Sleep(30000);
}

```

Figure 7: Performing some calculations using anonymous and standard procedural programming.



The results we find here are expected, but the extremity is still a bit shocking. Because of the work involved to setup a delegate, it is definitely a given that it should take more time. What we see here in this initial test is that using anonymous programming methods incur quite a bit of startup overhead, causing this simple function call to take over 50 times as long to complete.

The MSIL generated by this program when disassembled using ILDASM sheds some light on what is happening. An abridged version of the resultant MSIL is shown in Figure 8. Loads, stores, and other commonplace operations such as loop condition checking have been omitted for the sake of space.

What we are left with are the delegate setup portions mentioned herein. Each process is separated by a horizontal line. The full figure appears in Appendix A for

reference purposes. What we see is that the first method which uses outer variables produces code that duplicates the variable **i** for use internally by the delegate call (note the use of the `dup` instruction). We do not see the same behavior when we setup a delegate chain in our second process since it copies the variable to a local before passing it into the anonymous method. This avoids the problem of having to create instance variable copies for use within the anonymous method. This means that either method requires some form of copying. It's up to the programmer to avoid doing it poorly.

The first process also generates and loops over a larger amount of code, which helps explain why it takes the longest of our 3 styles. One of the biggest things to note with this method is that it calls the **Invoke()** method more than once. This is necessary to avoid printing incorrect values, as stated in Figure 7. While it gives us the desired output, it does not give us the desired performance. Despite this, it was very easy to write; on the surface it is not much different than our second process.

The second process demonstrates the better way to achieve the same result. Because we have created a delegate chain, we only call the **Invoke()** method one single

time. The entire delegate chain is then executed properly. This means that the only real overhead we incur here is setting up the delegate in the standard way.

The third process demonstrates the standard procedural way of achieving the desired output. We see here that there is little MSIL generated. It is fairly straightforward, as expected from something as simple as a for-loop. We skip the overhead of delegates and simply execute our instructions.

```

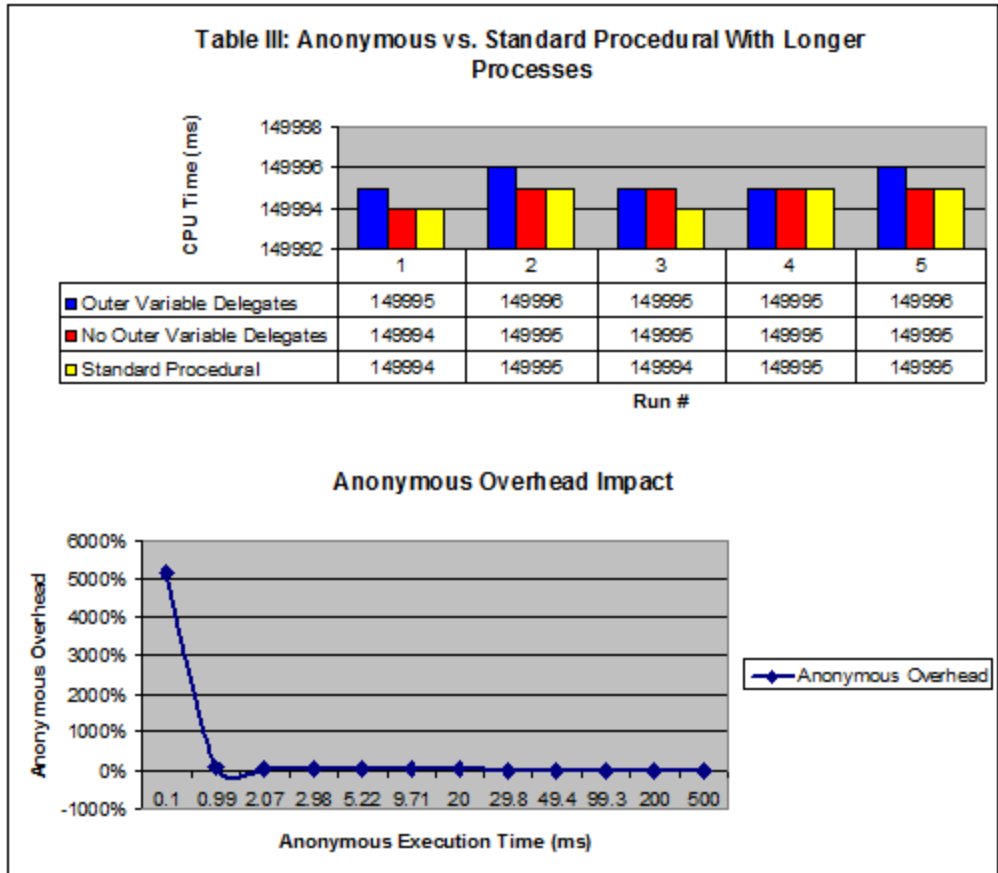
IL_0019: newobj      instance void tests/Program/'<>c__DisplayClass3':::ctor()
IL_001e: stloc.s      'CS$<>8_locals4'
IL_0020: ldloc.s      'CS$<>8_locals4'
IL_0022: ldc.i4.1
IL_0023: stfld        int32 tests/Program/'<>c__DisplayClass3':::i
IL_0028: br.s        IL_0059
IL_002b: ldloc.3
IL_002c: brtrue.s    IL_003e
IL_002e: ldloc.s      'CS$<>8_locals4'
IL_0030: ldftn        instance void tests/Program/'<>c__DisplayClass3':::<Main>b__0'()
IL_0036: newobj      instance void tests/Program/MyDelegate:::ctor(object, native int)
IL_003b: stloc.3
IL_003c: br.s        IL_003e
IL_0043: callvirt     instance void tests/Program/MyDelegate:::Invoke()
IL_004a: ldloc.s      'CS$<>8_locals4'
IL_004c: dup
IL_004d: ldfld        int32 tests/Program/'<>c__DisplayClass3':::i
IL_0052: ldc.i4.1
IL_0053: add
IL_0054: stfld        int32 tests/Program/'<>c__DisplayClass3':::i
IL_0059: ldloc.s      'CS$<>8_locals4'
IL_005b: ldfld        int32 tests/Program/'<>c__DisplayClass3':::i
-----
IL_0098: stloc.s      d2
IL_009a: ldc.i4.1
IL_009b: stloc.s      i
IL_009d: br.s        IL_00d6
IL_009f: newobj      instance void tests/Program/'<>c__DisplayClass5':::ctor()
IL_00a4: stloc.s      'CS$<>8_locals6'
IL_00a7: ldloc.s      'CS$<>8_locals6'
IL_00a9: ldloc.s      i
IL_00ab: stfld        int32 tests/Program/'<>c__DisplayClass5':::k
IL_00b0: ldloc.s      'CS$<>8_locals6'
IL_00b2: ldftn        instance void tests/Program/'<>c__DisplayClass5':::<Main>b__1'()
IL_00b8: newobj      instance void tests/Program/MyDelegate2:::ctor(object, native int)
IL_00bd: stloc.s      V_7
IL_00bf: ldloc.s      d2
IL_00c1: ldloc.s      V_7
IL_00c3: call        class [mscorlib]System.Delegate [mscorlib]System.Delegate:::Combine(class
      [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
IL_00c8: castclass   tests/Program/MyDelegate2
IL_00cd: stloc.s      d2
IL_00d0: ldloc.s      i
IL_00d2: ldc.i4.1
IL_00d3: add
IL_00d4: stloc.s      i
IL_00d6: ldloc.s      i
IL_00e2: brtrue.s    IL_009f
IL_00e4: ldloc.s      d2
IL_00e6: callvirt     instance void tests/Program/MyDelegate2:::Invoke()
IL_0117: ldc.i4.1
IL_0118: stloc.s      i
IL_011a: br.s        IL_012c
IL_011d: ldloc.s      i
-----
IL_011f: call        void tests/Program:::doABunchOfStuff(int32)
IL_0126: ldloc.s      i
IL_0128: ldc.i4.1
IL_0129: add
IL_012a: stloc.s      i
IL_012c: ldloc.s      i

```

Figure 8: Abridged MSIL disassembly

Even though anonymous methods, when setup optimally, still incur a performance hit due to setting up the delegate class and combining delegates when requested, it is still possible that their use should be encouraged. In this small example we see that the performance difference is quite great. However, if the setup performance hit of the delegate remains constant, we may see that this performance hit stops mattering when we begin performing computations and handling larger scale processes. The reason for this line of thinking is that currently, we are dealing with fractions of a millisecond. The performances of these 3 processes are different relative to each other, but in reality, may not be that different at all.

To test this idea out, we will again use the same 3 processes. This time however, we will need to simulate a longer computation than what is already done. This will be accomplished by having the function sleep for thirty seconds per run. The resultant test takes roughly seven and a half minutes to complete, with about two and a half minutes going to each of the three processes we have setup. This should be more than sufficient when compared to modern software which typically takes care of its processes in under a minute.



We see here in Table III that after sleeping for 30 seconds per function call, that there is little difference in the performance of these processes. Even the poorly executed anonymous method's performance does not appear different enough to matter. We even see that the anonymous approach often operates as fast as the standard procedural approach.

The accompanying line graph within Table III also shows the general impact of anonymous method overhead when used. Initially we see a terrible overhead impact much like we saw in Table II. However, we quickly see that as

the execution time increases steadily from .1ms to 1ms and beyond, that the overhead impact decreases significantly, eventually approaching 0%.

What this means is that as computation time increases, the impact of an anonymous methods setup decreases. The setup of the anonymous method does remain fairly constant, so it will always take roughly the same amount of time to setup, even if the function itself takes a long time to execute.

This of course means that anonymous methods are indeed a great tool to use within a project and should most certainly be used whenever a programmer needs or wishes to use one. The low impact of their overhead and heightened design possibilities (callbacks and event driven programming) make anonymous methods quite versatile. The overhead penalty incurred would only have a detrimental effect to programs which require operations that operate at the fraction of a millisecond level. Any program requiring very precise, almost instantaneous execution would generally want to avoid usage of anonymous methods. On the other hand, any program that is higher level and contains many different modules and design elements would most certainly benefit from the elegant flow that an anonymous method can provide.

2.3.2 Lambdas

Lambdas are a bit of a different case than standard anonymous programming. They are typically used in conjunction with LINQ extension methods and can operate as arguments to these methods. We will not need to examine the anonymous method setup as done previously, since lambdas operate the same in that regard and merely operate as a sort of short hand, much like **Action** and **Func**.

What we will instead be examining is a lambda's performance when used in place of standard procedural ways of programming.

Our example in Figure 9 creates a list of 2,000,006 strings. We have placed various book titles within the list at the beginning, middle, and end. What we then do is search for some of the titles using lambdas, and again using standard procedural programming.

The results in Table IV show that lambdas can be quite a silent killer to program efficiency if not used correctly. Further, as we can see in the coding example, it is fairly easy to set lambdas up in efficiently due to their short hand nature and streamlined appearance. Things that appear to be short and sweet may turn out to be long and sour.

```

public static void Main(string[] args)
{
    Stopwatch sw = new Stopwatch();

    books.Add("Neuromancer");
    for (int i = 0; i < 1000000; i++) books.Add(i.ToString());
    books.Add("Holomen");
    books.Add("Azure Bonds");
    for (int i = 0; i < 1000000; i++) books.Add(i.ToString());
    books.Add("Count Zero");
    books.Add("Assembly Lines");
    books.Add("Apple II Graphics");

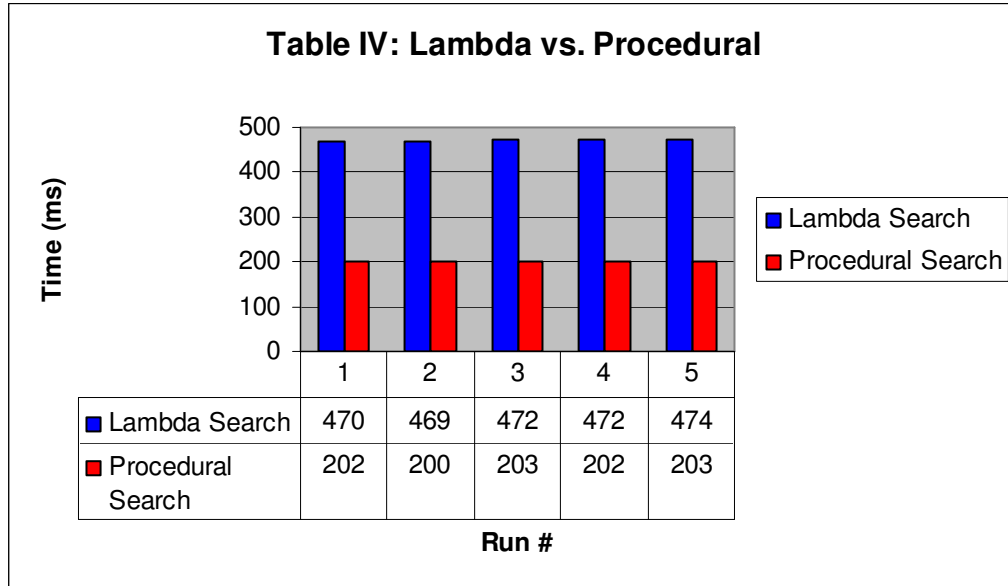
    sw.Reset();
    sw.Start();
    if (books.Exists(s => s.Equals("Neuromancer"))) processNeuromancer();
    if (books.Exists(s => s.Equals("Holomen"))) processHolomen();
    if (books.Exists(s => s.Equals("Azure Bonds"))) processAzureBonds();
    if (books.Exists(s => s.Equals("Count Zero"))) processCountZero();
    if (books.Exists(s => s.Equals("Assembly Lines"))) processAssemblyLines();
    if (books.Exists(s => s.Equals("Apple II Graphics"))) processAppleII();
    sw.Stop();

    Console.WriteLine("\nLambda search: {0}ms", sw.ElapsedMilliseconds);

    sw.Reset();
    sw.Start();
    foreach (string s in books)
    {
        if (s.Equals("Neuromancer")) processNeuromancer();
        else if (s.Equals("Holomen")) processHolomen();
        else if (s.Equals("Azure Bonds")) processAzureBonds();
        else if (s.Equals("Count Zero")) processCountZero();
        else if (s.Equals("Assembly Lines")) processAssemblyLines();
        else if (s.Equals("Apple II Graphics")) processAppleII();
    }
    sw.Stop();
    Console.WriteLine("\nProcedural search: {0}ms", sw.ElapsedMilliseconds);
}

```

Figure 9: Lambda and Procedural Searches



What our current, deceptive lambda search is doing in standard procedural form is shown in Figure 10. To elaborate, the lambda traverses the entire list in search of the string in question. Once it finds the string or reaches the end of the list, it returns. This means that each string we search for requires us to traverse the entire list again. In our example, this means our best case lambda search would be 21; the case where our 6 strings are the first six entries in the list, in the same order that we search for them. Our worst case on the other hand, is 12,000,036. This is what would happen if none of our search strings exist in the list and we are forced to run to the end each time. It should be noted that it is indeed possible for this lambda setup to outperform our procedural form, if, for example, our list was sorted and

we searched for everything in alphabetical order AND all of the books in question appear in the front portion of the list so that the accumulated traversals are less than the procedural's. However this is highly unlikely, and should also not be counted on. Searching should always be setup to accommodate the average and/or worst case.

```
foreach (string s in books){if (s.Equals("Neuromancer")) processNeuromancer();}
foreach (string s in books){if (s.Equals("Holomen")) processHolomen();}
foreach (string s in books){if (s.Equals("Azure Bonds")) processAzureBonds();}
foreach (string s in books){if (s.Equals("Count Zero")) processCountZero();}
foreach (string s in books){if (s.Equals("Assembly Lines")) processAssemblyLines();}
foreach (string s in books){if (s.Equals("Apple II Graphics")) processAppleII();}
```

Figure 10: Inefficient lambdas in procedural form

Our procedural version from Figure 9 will traverse the list 2,000,006 times, always. It is a guaranteed run time. It could be further optimized using Boolean variables to exit the for-loop if all books are found in order to achieve similar best case run times as the lambda.

This was just a simple test of lambdas, and we can already see that their easy to use nature can introduce some dangerous pitfalls that can go unnoticed. What this implies is that lambdas are best used in cases where they are simply taking the place of an already created anonymous method to improve readability. They should also be used if the situation truly requires a lambda. Cases such as this would include many of the calls to LINQ extension methods

much like what we have seen here, as you are not able to use them otherwise.

These cases often deal with performing queries on various data sources and may involve SQL like mechanisms, so it would be the programmer's duty to make sure the lambdas are being used in a manner that doesn't sacrifice efficiency. Examples of these kinds of lambda uses, along with their procedural counterparts are shown in Figure 11. We can see in these particular cases that the lambdas do provide cleaner looking code, and do in fact operate quicker, based off of the timings shown in Table V and Table VI.

The disassembly of the sample programs sheds some light on why this happens. Examining the MSIL in Figure 12 shows us the true power of using LINQ extension methods, complete with lambdas. What we see is that the anonymous method created via the lambda within each of our LINQ extension method calls is placed into memory (via `ldftn`). Then, the usual anonymous method setup occurs, and our LINQ extension method is called. It's as straightforward in the MSIL as it is in the C# code.

The amount of code generated for the LINQ search call is both shorter, and less involved. As we can see with the procedural search method, aside from the lengthier amount

of code, the compiler has also generated a try/catch statement for us to handle any exceptions that occur while performing the string comparison.

The LINQ style proves to be far more efficient and powerful for both our search case, *and* our sort case. The sort case is particularly better with LINQ because we don't incur the overhead of having our class inherit from **Comparable** and implement **CompareTo()** to perform our class sort. LINQ's internal workings allow it to quickly search *and* sort our data before our procedural style can even search the data in the first place. So, while the two Sort styles are about the same amount of code within MSIL, the LINQ method is far superior due to the setup of LINQ, and the fact that sorting lists composed of classes incurs some overhead that LINQ does not have to concern itself with. We did find that in some cases, LINQ operated slower. However, the difference was negligible, inconsistent, and only occurred with smaller datasets. These findings imply that the overhead of LINQ is similar to that of anonymous methods. Again meaning that it is a problem at first, but the scalability of it quickly makes it a powerful tool to leverage in practice.

```

// catalog setup excluded. It creates a list of 6,000,002 books, 1,500,002 of
// which are ScienceFiction, in the middle
sw.Reset();
sw.Start();
IEnumerable<Book> scifi = catalog.Where(g => g.genre == "Science Fiction");
sw.Stop();
Console.WriteLine("Lambda search took {0}. Length = {1}", sw.Elapsed, scifi.Count());

sw.Reset();
sw.Start();
List<Book> scifi2 = new List<Book>();
foreach (Book b in catalog)
{
    if (b.genre == "Science Fiction")
        scifi2.Add(b);
}
sw.Stop();
Console.WriteLine("Procedural search took {0}. Length = {1}.\n", sw.Elapsed, scifi2.Count);

sw.Reset();
sw.Start();
IEnumerable<Book> byName = scifi.OrderBy(book => book.title);
sw.Stop();
Console.WriteLine("Lambda sort took {0}.", sw.Elapsed);

sw.Reset();
sw.Start();
scifi2.Sort();
sw.Stop();
Console.WriteLine("Procedural sort took {0}.", sw.Elapsed);

```

Figure 11: Searching and Sorting with lambdas and procedural programming.

Table V: Lambda LINQ Search vs. Standard Procedural Search

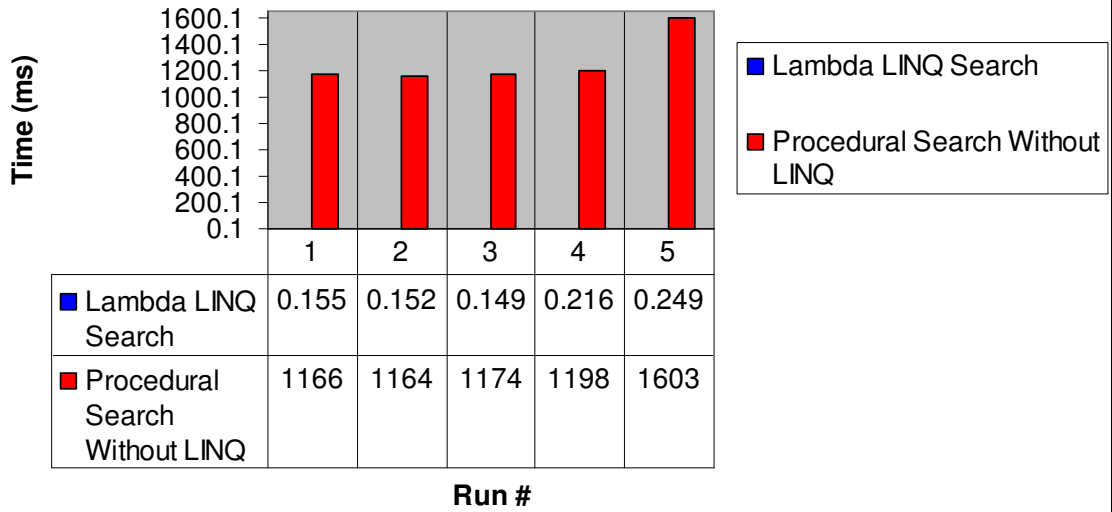
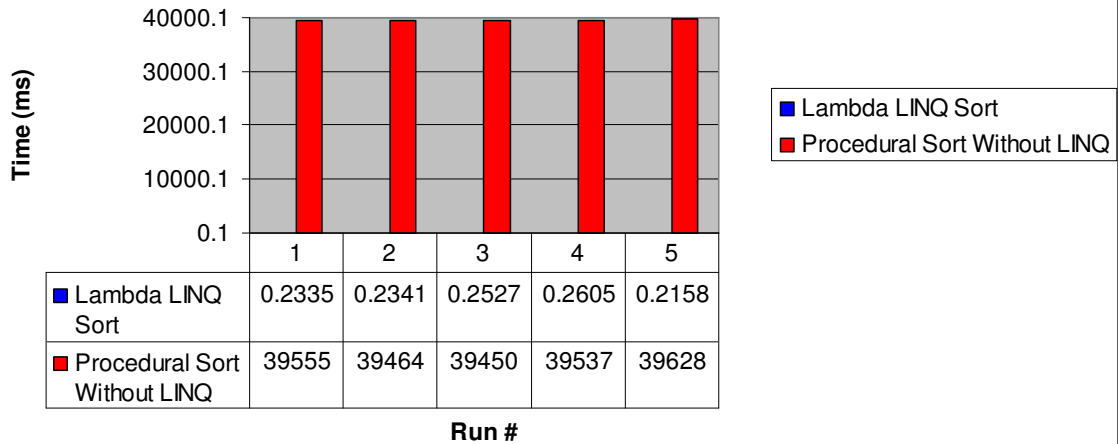


Table VI: Lambda LINQ Sort vs. Standard Procedural Sort



```

IL_0189: ldsfld      class [mscorlib]System.Func`2<class tests/Program/Book,bool>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_0191: ldftn      bool tests/Program::'<Main>b_0'(class tests/Program/Book)
IL_0197: newobj     instance void class [mscorlib]System.Func`2
<class tests/Program/Book,bool>::ctor(object, native int)
IL_019c: stsfld     class [mscorlib]System.Func`2<class tests/Program/Book,bool>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_01a1: br.s      IL_01a3
IL_01a3: ldsfld     class [mscorlib]System.Func`2<class tests/Program/Book,bool>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_01a8: call      class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>
[System.Core]System.Linq.Enumerable::Where<class tests/Program/Book>
(class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>,
class [mscorlib]System.Func`2<!!0,bool>)
IL_01ad: stloc.s   scifi
IL_01c6: ldloc.s   scifi
IL_01c8: call     int32 [System.Core]System.Linq.Enumerable::Count<class
tests/Program/Book>(class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>)
IL_01e6: newobj   instance void class [mscorlib]System.Collections.Generic.List`1
<class tests/Program/Book>::ctor()
IL_01eb: stloc.s   scifi2
IL_01ee: ldloc.1
IL_01ef: callvirt instance valuetype [mscorlib]System.Collections.Generic.List`1
/Enumerator<!0>class [mscorlib]System.Collections.Generic.List`1
<class tests/Program/Book>::GetEnumerator()
IL_01f4: stloc.s   CS$5$0001
IL_01f6: br.s     IL_0227
IL_01fa: call     instance !0 valuetype [mscorlib]
System.Collections.Generic.List`1/Enumerator<class tests/Program/Book>::get_Current()
IL_01ff: stloc.s   b
IL_0202: ldloc.s   b
IL_0204: ldfld     string tests/Program/Book::genre
IL_0209: ldstr     "Science Fiction"
IL_020e: call     bool [mscorlib]System.String::op_Equality(string, string)
IL_0213: ldc.i4.0
IL_0214: ceq
IL_0216: stloc.s   CS$4$0000
IL_0218: ldloc.s   CS$4$0000
IL_021a: brtrue.s  IL_0226
IL_021c: ldloc.s   scifi2
IL_021e: ldloc.s   b
IL_0220: callvirt instance void class [mscorlib]System.Collections.Generic.List`1
<class tests/Program/Book>::Add(!0)
IL_0229: call     instance bool valuetype [mscorlib]System.Collections.Generic.List`1
/Enumerator<class tests/Program/Book>::MoveNext()
IL_022e: stloc.s   CS$4$0000
IL_0230: ldloc.s   CS$4$0000
IL_0232: brtrue.s  IL_01f8
IL_0234: leave.s   IL_0245
IL_0238: constrained. valuetype [mscorlib]System.Collections.Generic.List
`1/Enumerator<class tests/Program/Book>
IL_023e: callvirt instance void [mscorlib]System.IDisposable::Dispose()
IL_025d: ldloc.s   scifi2
IL_025f: callvirt instance int32 class [mscorlib]System.Collections.Generic.List`1
<class tests/Program/Book>::get_Count()
IL_0264: box      [mscorlib]System.Int32
IL_0269: call     void [mscorlib]System.Console::WriteLine(string,object, object)
IL_027d: ldloc.s   scifi
IL_027f: ldsfld     class [mscorlib]System.Func`2<class tests/Program/Book,string>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_0284: brtrue.s  IL_0299
IL_0287: ldftn     string tests/Program::'<Main>b_1'(class tests/Program/Book)
IL_028d: newobj     instance void class [mscorlib]System.Func`2<class
tests/Program/Book,string>::ctor(object, native int)
IL_0292: stsfld     class [mscorlib]System.Func`2<class tests/Program/Book,string>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_0299: ldsfld     class [mscorlib]System.Func`2<class tests/Program/Book,string>
tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_029e: call     class [System.Core]System.Linq.IOrderedEnumerable`1<!!0>
[System.Core]System.Linq.Enumerable::OrderBy<class tests/Program/Book,string>(class
[mscorlib]System.Collections.Generic.IEnumerable`1<!!0>,!!1)
IL_02a3: stloc.s   byName
IL_02d0: ldloc.s   scifi2
IL_02d2: callvirt instance void class [mscorlib]System.Collections.Generic.List`1
<class tests/Program/Book>::Sort()

```

Figure 12: MSIL disassembly of searching and sorting with lambdas and procedural programming.

Again referring to both Tables V and VI, we see that the difference in run-times between the LINQ methods and the standard procedural approaches is quite radical. In order to figure out why, these LINQ extension methods must be viewed closer. Using .NET Reflector, we are able to investigate what goes on with LINQ extension methods, as shown in Figure 13. What we see is that the LINQ method **Where()** makes use of something interesting known as a **yield return**. We will cover this in detail in the next section. To put it simply for now, what it does is take advantage of the enumerable type within C#, which helps explain why our LINQ search went far faster than the rather brute force approach taken with procedural code that uses basic comparisons and builds a **List()** on the fly.

We also see that the **OrderBy()** method makes use of the **OrderedEnumerable** type, which essentially allows the framework itself to craft our sorted list for us as it is built. This is done instead of implementing a **CompareTo()** method and a standard **Sort()** function which as we see in Figure 14, does not perform very optimally for us. The work for **List.Sort()** is passed off to the Array class, and then to an **ArraySortHelper**, which ends up calling the C# implementation of the *quicksort algorithm*. By nature, this algorithm will run in $O(n \log n)$ time on average. When we

see larger portions of data, much like our example, this algorithm will run well when compared to other conventional sort algorithms. However, LINQ's **OrderBy()** method has the advantage of being setup to take advantage of **Enumerables**, and the rest of the C# framework. This allows it to create our new data structure quickly and efficiently, in a manner similar to that of a best time Insertion sort (which runs in $O(n)$ time). This proves it to be quite powerful and versatile.

```

internal static IEnumerable<T> Where<T>(this IEnumerable<T> enumerable, Func<T, bool> where)
{
    foreach (T iteratorVariable0 in enumerable)
    {
        if (where(iteratorVariable0))
        {
            yield return iteratorVariable0;
        }
    }
}

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>
    (this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)
{
    return new OrderedEnumerable<TSource, TKey>(source, keySelector, null, false);
}

internal class OrderedEnumerable<TElement, TKey> : OrderedEnumerable<TElement>
{
    // Fields
    internal IComparer<TKey> comparer;
    internal bool descending;
    internal Func<TElement, TKey> keySelector;
    internal OrderedEnumerable<TElement> parent;

    // Methods
    internal OrderedEnumerable(IEnumerable<TElement> source,
        Func<TElement, TKey> keySelector, IComparer<TKey> comparer, bool descending);

    internal override EnumerableSorter<TElement> GetEnumerableSorter(EnumerableSorter<TElement> next);
}

```

Figure 13: Reflecting Upon LINQ Extension Methods


```

internal static void QuickSort(T[] keys, int left, int right, IComparer<T> comparer)
{
    do
    {
        int a = left;
        int b = right;
        int num3 = a + ((b - a) >> 1);
        ArraySortHelper<T>.SwapIfGreaterWithItems(keys, comparer, a, num3);
        ArraySortHelper<T>.SwapIfGreaterWithItems(keys, comparer, a, b);
        ArraySortHelper<T>.SwapIfGreaterWithItems(keys, comparer, num3, b);
        T y = keys[num3];
        do
        {
            while (comparer.Compare(keys[a], y) < 0)
            {
                a++;
            }
            while (comparer.Compare(y, keys[b]) < 0)
            {
                b--;
            }
            if (a > b)
            {
                break;
            }
            if (a < b)
            {
                T local2 = keys[a];
                keys[a] = keys[b];
                keys[b] = local2;
            }
            a++;
            b--;
        }
        while (a <= b);
        if ((b - left) <= (right - a))
        {
            if (left < b)
            {
                ArraySortHelper<T>.QuickSort(keys, left, b, comparer);
            }
            left = a;
        }
        else
        {
            if (a < right)
            {
                ArraySortHelper<T>.QuickSort(keys, a, right, comparer);
            }
            right = b;
        }
    }
    while (left < right);
}

```

Figure 14: List.Sort() 's Behind the Scenes Footage

With this, we see that C# provides us with many great facilities to design very robust, elegant programs. We also see that the efficiency is not at great risk when taking advantage of these features.

The next frontier is dynamic programming.

CHAPTER III

DYNAMIC PROGRAMMING

3.1 The Common Language Runtime

The Common Language Runtime is the .NET Framework's means to allow for portable code while using C#. It is the virtual machine that the .NET framework uses. This is where Just-In-Time (JIT) compilation takes place. This is also where some of the other features of the .NET framework take place, including memory management and type-safety mechanisms. The CLR is essentially the manager of this managed language.

The CLR provides some features inherent to managed programming such as type safety and memory management AKA "garbage collection". These features are imposed on a programmer using C# no matter what. So, because we are unable to use C# without these features, we will not be examining them further. Suffice it to say, the garbage

collection and dispose patterns employed by the CLR are more than adequate for managed programming. Further, the type safety mechanisms in place, much like what we have seen put into place when we make use of delegates and anonymous programming, are part of what makes C# as effective as it has become. Two key features of the CLR however, are being brought under examination. They are the JIT compiler, and Reflection

Our code written in C# is translated by the compiler into MSIL, and from there, the JIT compiler transforms it into native code for the target architecture. The question becomes now, is this JIT compilation approach "good enough" to rely on? Or, should a programmer be wary that they still need to massage their code by hand to get the efficiency they desire?

3.1.1 Just In Time Compilation

Of all of the features within the CLR, JIT compilation has drawn much attention, as this is the main force of optimization with managed code. The JIT compiler is the last piece of optimization before the CLR completely translates MSIL into native code. This means that if this compiler is not very optimal, the end-result on each platform will not be very good. It may also introduce radically different performance from one architecture to the other. This would ruin the concept of portability.

One of the biggest hurdles to overcome with JIT compilation is that it runs under time constraints. It is unable to make use of more conventional means of optimization since it is done "Just in Time". Obviously, the optimization cannot be done just in time if it takes a long amount of time to complete.

Sasha Goldshtein's "JIT Optimizations" article examines some of the optimization strategies used by the CLR, mainly focusing on using method inlining and frequency analysis. He first points out the simple fact that range check optimizations within loops can be broken quite easily. This is our first sign that it may still be up to the programmer to perform their own optimizations as much as possible.

However, he then points out that methods are inlined if they are 32 bytes or less in length, do not contain complex branching logic, and do not use exception handling. What this then means is that larger functions will not be inlined. This is a normal occurrence. In C/C++, only small, tightly knit functions are usually inlined into the code. Essentially, a function should only be inlined if the function can complete its processes faster than the overhead for calling that same function can. Otherwise, there is little point to it.

The function inlining optimizations are interesting because of the fact that Goldshtein points out that it is theoretically impossible to perfectly inline virtual method calls because the JIT does not inline interface method calls. Instead, it performs an optimization that does not use naïve interface method dispatching.

He then goes on to examine flow analysis and frequency analysis to explore JIT optimizations. The end conclusion is that the frequency of method calls and the resultant optimizations have little impact on performance. One could interleave various method calls, or call them sequentially, and it would have little difference because of the way the JIT compiler decides to optimize.

This does not raise much concern, as what it means really is that one programmers design patterns will not be drastically different than another's. The JIT compiler and its optimizations seem to have created a middle ground for programming within C#.

The real question here is are there other smaller optimizations that could be made that cry back to the assembly language days where hand massaging code could produce far superior code. We do not have the luxury of inlining assembly language within a C# program as one would be able to with C/C++, but we do have the option to emit code on the fly.

This, however, is not what will be experimented with, as it is not quite the same thing. Emitting code within C# on the fly is more of a dynamic process than an optimizing process, as you are simply emitting classes, methods, and other things on the fly. You are basically telling the C# compiler what sort of MSIL to generate, and are still forced to comply with all of the conventions of C# in that regard.

What we will instead be investigating is if careful code setup can produce noticeably faster code for us.

The first test is to see if copying an array's length to a local variable to use for loop condition checking is

faster than using the array's length property with each check, as shown in Figure 15.

```

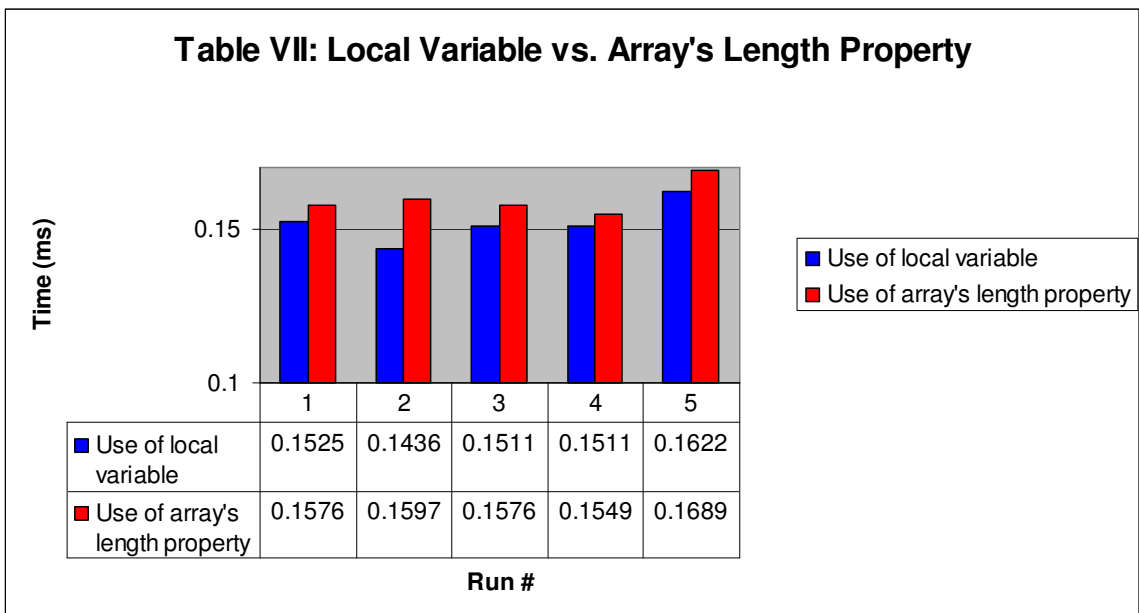
int[] test = new int[50000];
int len = test.Length;

for (int i = 0; i < test.Length; i++)
{
}
for (int i = 0; i < len; i++)
{
}

```

Figure 15: local/non local lengths

The results of this test show that there is little difference between the two, so it generally makes more sense to just use the built in property of the array in order to keep cleaner code. Obvious cases where this would not apply are the cases where you need the length outside of the loop as well. Table VII shows the results of this test, and demonstrate the negligible differences of the two approaches.



Doing this again with a list instead of an array yields different results. Figure 16 shows the approach taken, and Table VIII shows the result. This time, we see initially that using List.Count takes about twice as long to complete each time. Disassembling this, we see that it is because Count is retrieved via a function call, rather than retrieving a variable. This makes sense due to the fact that a list's size is able to be changed and thus must have some sort of way to iterate and count the list.

Therefore, the amount of work done varies depending on the size of the list in question. As the line graph in Table VIII shows, the overhead is always higher using the Count property than if we were to use a local variable, and its best case scenario seems to fall within the 87-88% range starting at a length of 15,000. However, the result is similar to the anonymous method investigation in that the overhead involved is outweighed significantly by convenience whenever the complexity of the program as a whole increases. Even with large lists, the time it takes to process the loop condition information is under one second and tends to take about 400ms in our largest list size case that reached the limit of the memory on our test system.

This amount of work will quickly be made irrelevant by the actual computations that will take place in the loop in question. In our largest case, any operation that takes at least 401ms will be taking longer than the overhead of our property usage, thus making it become more and more worth it to use as we approach 1 second long operations, or even longer.

It is implied now that in order for it to be the most useful to use a local variable instead of the Count property, we must be working with very small datasets. In Table VIII, we see this is at a size of 1500, as this is when our overhead for using the Count property exceeds 100. Even so, the performance gain at this level would only be beneficial if we are working with very time sensitive applications. This is because we will still only be gaining fractions of a millisecond. Because of this, it is likely that the speed difference between both styles of coding will be unnoticed, so it makes more sense to use the Count property. This property removes the need to manage a variable whose main purpose is to act as our for-loop condition exit value. So again, the only time it would make a great deal of sense to use a local variable is if we intend to use the length outside of the loop as well.

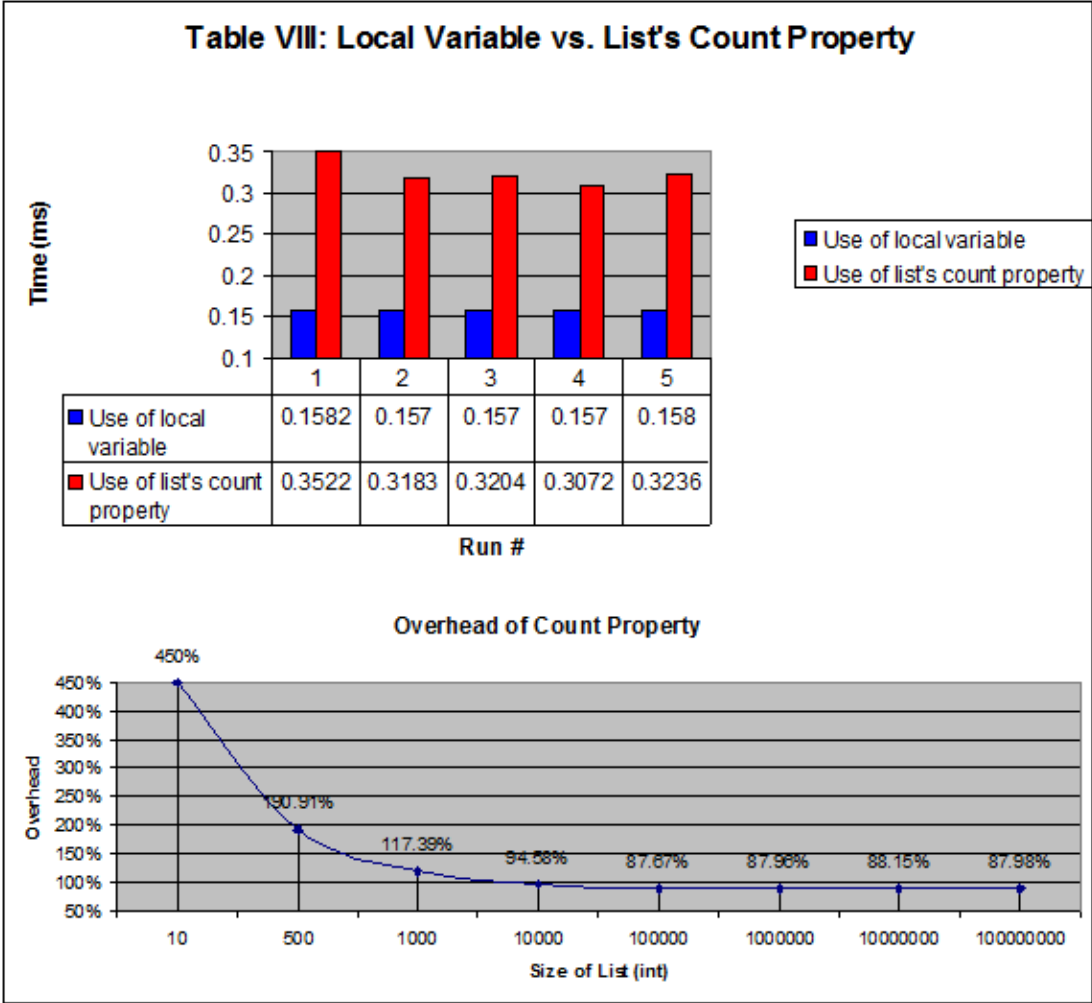
```

int len = test.Count;

for (int i = 0; i < test.Count; i++)
{
}
for (int i = 0; i < len; i++)
{
}

```

Figure 16: Local variable versus Count property



This now shows that most “hand rolled” optimizations brought on by being mindful of the code will not cause much benefit to the end result. Because the language as a whole is many levels above the bare metal, the more prevalent optimizations are not possible.

More on the futility of hand done optimizations comes from Jeffery Richter's book "CLR via C#". Early on in this book, he states that the CLR's use of the JIT compiler produces very efficient code. He even challenges readers to try it for themselves, which is what we have done here. One of the key reasons for the JIT compilation being so powerful is that it knows a lot of things ahead of time, including CPU architecture. This allows it to take advantage of any possible nuances certain architectures may have. These specific advantages are far more important and vital to optimized code than the various tricks we may attempt on our own.

Significant research has gone into the topic of JIT compilation, mainly on the topic of trace-based JIT optimizations as a means to further improve JIT compilation. What it does is take advantage of run-time profiling to optimize the most frequently executed paths within the code while also providing a means to bail out if this path becomes invalid.

This was demonstrated in Dr. Andreas Gal's dissertation entitled "Efficient Bytecode Verification and Compilation in a Virtual Machine". The work done in this paper demonstrates that trace based dynamic compilation can produce versatile results that further reduce the benefit

of using hand-done optimizations. This agrees with what we've seen previously through our own experimentation. Gal's trace-tree based dynamic compiler also managed to outperform traditional Java Virtual Machines (JVMs) that were used during testing. The only competition came from HotSpot. However, HotSpot was not created with embedded systems in mind whereas Gals' trace-tree was. This means that HotSpot may compete in terms of speed, but cannot compete in terms of file size and memory consumption.

In a paper entitled "The Essence of Compiling With Traces" by Shu-yu Guo and Jens Palsberg, we see further exploration of the same trace based compilation concept. This time, however, we see the investigation of sound optimizations with trace compilation. The paper details ways to determine if traces are correct. In order for them to be correct, they must "do the same thing" as the original code. The conclusion that follows is that by using bisimulation to overcome the explicit definitions of JIT compilation and using confluence to maintain continuity with operation correctness, one can create sound optimizations with trace based JIT compilation.

This research provides a great foundation for further improvements to JIT compilation in the future. It also helps show that in its current state, JIT compilation is

versatile enough that programmers do not need to concern themselves with trying to coax the compiler into doing more efficient things for them; it's already taken care of. In order to obtain more robust results from JIT compilation, one would need to look outside the realm of their own code and explore improving the JIT compilation at its very core using methods shown here, or perhaps by a new method altogether.

3.1.2 Reflection

In *Effective C#*, Bill Wagner briefly mentions a process known as reflection with regards to getting the name of a calling method. While mentioning it, he states that it greatly simplifies tasks, but also states that it is an expensive process. Simple but expensive is a bit of a red flag when efficiency is the question. So, we need to find out just how expensive reflection is. Is the simplification of code worth the expensiveness of reflection? Also, what exactly is expensive about it?

The MSDN states that reflection is useful for the following:

- Accessing attributes in your programs meta data
- Examining and instantiating types in an assembly
- Building new types at runtime
- Performing late binding

Moreover, we see that reflection can be used to perform some things that would not normally be possible without reflection. What is demonstrated in Figure 17 is that by using reflection, one can access a private member within a class. This is not recommended or encouraged, but it is indeed possible. Without reflection, there is no way to access this private method; reflection can be used to

bypass some of C#'s rules. This is dangerous. However, some things that are dangerous in programming do have useful applications.

```
class Program
{
    class Tester
    {
        int number = 88;
        public void Inform()
        {
            Console.WriteLine("Hi, my value is " + number);
        }

        private void Increment()
        {
            number++;
        }
    }

    static void Main(string[] args)
    {
        Tester tester = new Tester();
        tester.Inform();
        tester.GetType().GetMethod("Increment", System.Reflection.BindingFlags.NonPublic
            | System.Reflection.BindingFlags.Instance
            | System.Reflection.BindingFlags.InvokeMethod).Invoke(tester, new object[] { });
        tester.Inform();
    }
}
```

Figure 17: Using reflection to access private methods.

We see here in this simple example that one can extract methods off of an instance and reflect upon them dynamically to invoke different pieces of a class. This shows part of the real power and benefit to using reflection.

However, the power is not free. It comes with significant overhead, and the use of reflection is notoriously referred to as being an expensive (as stated by Wagner) drain on performance. Using numerical data borrowed by Eric McMullen in his article "Get Drunk on the

Power of Reflection.Emit", we see that the use of reflection is indeed slower than the standard new operator as far as creating objects per second: 708,160 for Reflection and 3,160,493 for the new operator. This is a significant difference.

It should be noted that reflection often incurs a one time performance hit at load-time. After this, the results are typically cached for fast retrieval. This means you can generate many things at run time and create very dynamic code that only incurs a performance hit once.

Research has shown that the Reflection debate is split in half. There are those who agree that it is bad news and should be avoided as much as possible, and there are those who argue that it is not as bad as it seems, and it opens up many possibilities for dynamic code with minimal overhead. There are many online discussions that debate the benefits of using them.

We are of the thought that reflection is not something that should be used constantly, as it can lead to difficult to maintain code that also runs very poorly.

What we have discovered here is that reflection is a difficult process to spell out specifically. Because of its dynamic nature, reflection can be applied to many different scenarios and it is difficult to discern when it

should actually be used. There is *always* an efficiency hit implied whenever reflection is used, and you can't coax reflection into performing on par with conventional methods as there aren't any to compare it to. So, determining when to use it and when to shy away from it is up to the programmer's discretion based off of their current situations and goals. If you can afford the couple of seconds of reflecting, it is probably worth it to use it if you gain a lot of flexibility from the use of it. Again going back to McMullen's numerical data, we see that the amount of objects created per second for reflection, while slower than the standard new operator, is not really an awful number. Being able to create that many objects per second is most certainly more than enough for an application to perform its tasks.

This is very similar to the anonymous method data we created previously. We are essentially dealing with fractions of a second in performance difference. This is again something that is not very prevalent. The true performance bottlenecks of an application are more likely to rear their heads elsewhere, most likely in the form of I/O or network access. This sort of bottleneck is out of the hands of the C# programmer.

Figure 18 shows an example of reflection NOT being used to return various handlers, and Figure 19 shows the same sort of process via reflection. The basic premise of this example with respect to reflection is that we create a small database of possible handlers and store them in a dictionary by their string names. Whenever we need one, we check the dictionary for the existence of that handler, and dynamically build and return it out to the user.

Reflection is certainly not required for this, and the code could be simplified significantly by simply returning the handlers via the new operator rather than building the dictionary of handler types that get dynamically generated (as shown in Figure 18). However, the use of reflection does allow us to create a fairly versatile database of handlers with a minimal amount of code. It may seem less straight forward because it is dynamic and thus will incur the normal reflection runtime penalties, but it will provide us with the means to add or remove handlers with ease. We now have a one-stop shop for any handler. If it turns out that we need to change how a handler is used (perhaps the signature of the constructor needs changed), we only need to modify the reflection portion one single time. With the approach taken in Figure 18, we would have

to change each and every one of the handlers function calls.

On the other hand, the standard returning of handlers via the new operator approach allows us to simply check the `DisplayType` in question, and return the appropriate handler. We will be able to debug this code in the standard way, and the code itself is more straightforward. However, because it is not dynamically generated, if we ever change the way we deal with handlers being returned, we will need to change every place in which the new operator appears in. Reflection would only require us to change it in the spot where we dynamically generate it since we are retrieving the handler by a simple string name and letting reflection do the real work for us.

```

namespace Test
{
    internal static class HandlerFactory
    {
        internal static Handler GetHandler()
        {
            return new Test.DefaultHandler();
        }

        internal static Handler GetHandler(TestFile ft)
        {
            return getHandler(ft);
        }

        private static Handler getHandler(TestFile ft)
        {
            DisplayType displayType = ft.FileType.DisplayType;

            switch (displayType)
            {
                case DisplayType.Text:
                {
                    return new Test.TextHandler();
                }
                case DisplayType.Image:
                {
                    return new Test.ImageHandler();
                }
                case DisplayType.HTML:
                {
                    return new Test.HTMLHandler();
                }
                default:
                {
                    return new Test.DefaultHandler();
                }
            }
        }
    }
}

```

Figure 18: Retrieving handlers without reflection.

```

namespace Test
{
    internal static class HandlerFactory
    {
        internal static Handler GetHandler(){return HandlerRegistrar.Default;}

        internal static Handler GetHandler(TestFile ft)
        {
            string name = findHandler(ft);
            Handler handler = HandlerRegistrar.Instance.GetHandler(name);
            return handler;
        }

        private static string findHandler(TestFile ft)
        {
            FileTypes fileType = ft.FileType.FileTypeID;
            DisplayType displayType = ft.FileType.DisplayType;

            switch (displayType)
            {
                case DisplayType.Text:
                {
                    return HandlerRegistrar.Text;
                }
                case DisplayType.Image:
                {
                    return HandlerRegistrar.Image;
                }
                case DisplayType.HTML:
                {
                    return HandlerRegistrar.HTML
                }
                default:
                {
                    return HandlerRegistrar.Default;
                }
            }
        }
    }
}

internal interface IHandlerFactory{Handler GetHandler();}

internal sealed class HandlerRegistrar
{
    private static readonly MethodAttributes methodAttributes = MethodAttributes.Public |
    MethodAttributes.Virtual | MethodAttributes.NewSlot | MethodAttributes.Final
    | MethodAttributes.HideBySig;
    internal const string Default = "Test.Default";
    internal const string Text = "Test.Text";
    internal const string Image = "Test.Image";
    internal const string HTML = "Test.HTML";
}

```

Figure 19: Dynamically creating handlers using reflection.

```

internal static HandlerRegistrar Instance
{
    get{return new HandlerRegistrar();}
}

private AssemblyBuilder _assemblyBuilder;
private ModuleBuilder _moduleBuilder;
private Dictionary<string, IHandlerFactory> _factories = new Dictionary<string, IHandlerFactory>();

private HandlerRegistrar() { }

internal ModuleBuilder ModuleBuilder
{
    get
    {
        if (_moduleBuilder == null)
        {
            AssemblyName name = new AssemblyName();
            name.Name = "HandlerFactory";
            _assemblyBuilder = AppDomain.CurrentDomain.DefineDynamicAssembly(name,
                AssemblyBuilderAccess.RunAndSave);

            _moduleBuilder = _assemblyBuilder.DefineDynamicModule("HandlerFactory.dll"
                , "HandlerFactory.dll");
        }
        return _moduleBuilder;
    }
}

internal Handler GetHandler(string name)
{
    IHandlerFactory factory;
    if (! factories.TryGetValue(name, out factory)){
        throw new InvalidOperationException(name + " has no HandlerFactory. Check HandlerRegistrar.");
    }
    return factory.GetHandler();
}

public override void Register(string dataSourceName)
{
    foreach (string s in Types)
    {
        Type type = GetType(s);
        if (type != null)
        {
            IHandlerFactory factory = buildHandlerFactory(type);
            _factories.Add(type.FullName, factory);
        }
    }
}

```

Figure 19 (continued): Dynamically creating handlers using reflection.

```

internal override string Name{get{return "Handlers";}}
private IHandlerFactory buildHandlerFactory(Type handlerToBuild)
{
    if (!handlerToBuild.IsPublic && (handlerToBuild.Assembly != typeof(Handler).Assembly))
    {
        throw new InvalidOperationException(handlerToBuild.FullName + " is not public.");
    }

    ConstructorInfo ci = handlerToBuild.GetConstructor(BindingFlags.Instance | BindingFlags.Public,
                                                         null, new Type[] { }, new ParameterModifier[] { });
    if (ci == null)
    {
        throw new InvalidOperationException(handlerToBuild.FullName +
                                           " should have a public empty constructor.");
    }

    string className = handlerToBuild.FullName.Replace('.', '_') + "Factory";

    TypeBuilder typeBuilder = ModuleBuilder.DefineType(className, TypeAttributes.Public
| TypeAttributes.Class | TypeAttributes.BeforeFieldInit, typeof(object),
    new Type[] { typeof(IHandlerFactory) });

    ConstructorInfo objCtor = typeof(object).GetConstructor(new Type[0]);
    ConstructorBuilder pointCtor = typeBuilder.DefineConstructor(MethodAttributes.Public
| MethodAttributes.HideBySig, CallingConventions.Standard, new Type[0]);
    ILGenerator ctorIL = pointCtor.GetILGenerator();
    ctorIL.Emit(OpCodes.Ldarg_0);
    ctorIL.Emit(OpCodes.Call, objCtor);
    ctorIL.Emit(OpCodes.Ret);

    MethodBuilder mb = typeBuilder.DefineMethod("GetHandler", methodAttributes,
                                                typeof(Handler), new Type[] { });

    ILGenerator il = mb.GetILGenerator();
    il.Emit(OpCodes.Nop);
    il.Emit(OpCodes.Newobj, ci);
    il.Emit(OpCodes.Ret);

    Type type = typeBuilder.CreateType();
    return Activator.CreateInstance(type) as IHandlerFactory;
}

protected override IEnumerable<string> Types
{
    get
    {
        yield return Default + ", Test";
        yield return Text + ", Test";
        yield return Image + ", Test";
        yield return HTML + ", Test";
    }
}
}
}

```

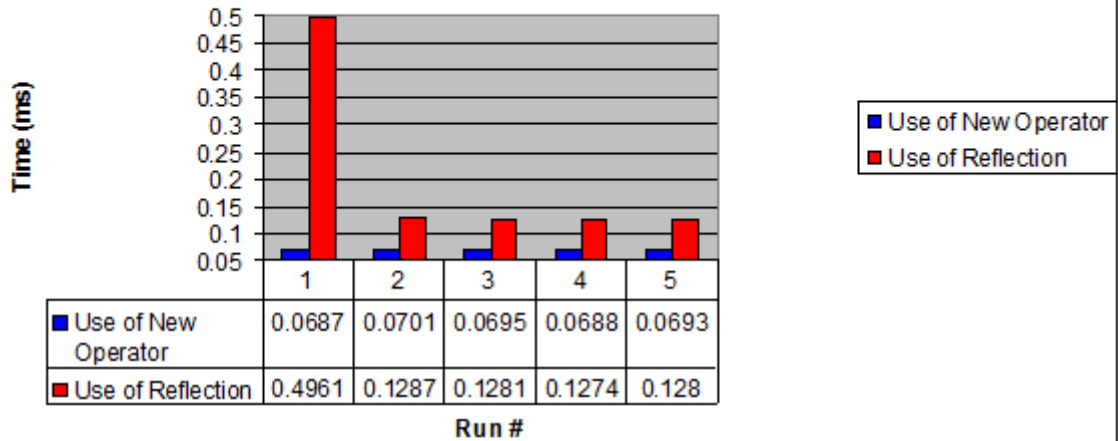
Figure 19 (continued): Dynamically creating handlers using reflection.

Upon utilizing the two versions, we can see in Table IX that the time for reflection is somewhat high in the first pass as it sets itself up, and each subsequent pass is significantly faster. It is still never as fast as the new operator is, but we can see that reflection is allowing

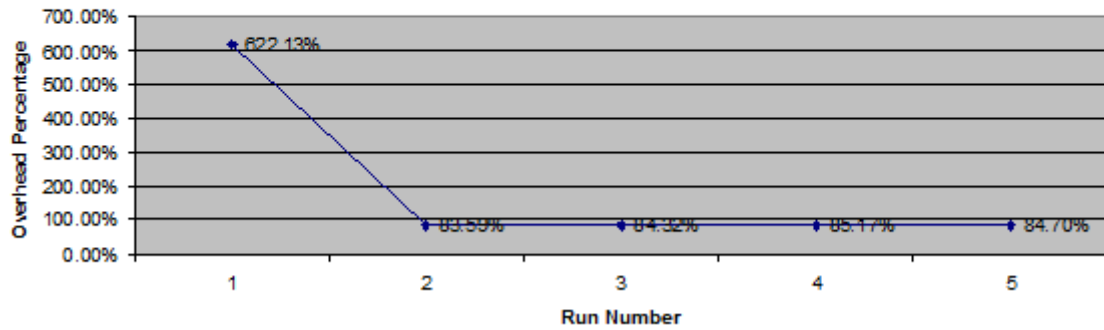
the handlers created to be cached and reused without having to regenerate them each time. The new operator is roughly 50% faster than reflection once reflection does its initial run. However, we yet again see that this is all fractions of a second, so it is very likely that the scenario in question and many others like it will allow for reflection to be used. The design benefits gained from reflection can outweigh the performance hit that is always implied and incurred with reflection, especially when the performance hit is again something that is not even discernable to the programmer or user of a program.

Because a programmer can know ahead of time that reflection causes this kind of performance impact, they may also be able to plan for and expect it in the design phase of their program so that it does not come as a surprise to them later on. Table IX demonstrates this expected overhead with a line graph. This graph serves to show the sort of overhead a programmer should expect when using reflection to dynamically generate types versus doing it with the standard new operator. It is certainly steep at first, and improves significantly with each subsequent run. We still see here that the overhead is costly, so this should be planned for if reflection is to be used.

Table IX: New Operator vs. Reflection



Reflection Overhead



Reflection as a whole has turned out to be a very flexible, very detailed feature of C#. Its power is something that is left up to the user's creativity to really take full advantage of it. One thing that was noted was the use of the Yield operator within the reflection example. It is also a new device of C#, so it needs to be examined.

3.2 Iterators Via Yield

While experimenting with reflection, we noted the use of a new operator called "yield". Researching this via the MSDN and the book "Accelerated C# 2008" by Trey Nash, we discovered that this is commonly used within iterator blocks. What it allows us to do is to step through a collection such as an **IEnumerable** one at a time, returning the item, and keeping track of where it left off for the next pass. This can be beneficial when looping through potentially large sets of data.

With respect to efficiency, most signs seem to point to it being more efficient with regard to iterating over large lists that you do not intend to fully traverse. This is beneficial to the memory consumption of your program since entire lists will not need to be stored in memory in order to be dealt with. Instead, you will be returned each piece as you require it. It is also beneficial in the event that you break out of a loop before building a list since you would be short cutting out of iterating over the entire list.

The best way to verify this is to experiment. What we have done is demonstrate the benefit of using yield return rather than a standard list in Figure 20. We see that using the list style causes us to build up the entire list

before being able to even process it. Since we are simply iterating over the list to find a value, we find that it is much more efficient to use yield return, as we may not even care about the later portions of the list.

```
class Program
{
    static void Main()
    {
        List<double> powers = getPowers(2, 32);
        foreach (double d in powers)
        {
            if (d == 65536.0) break;
        }

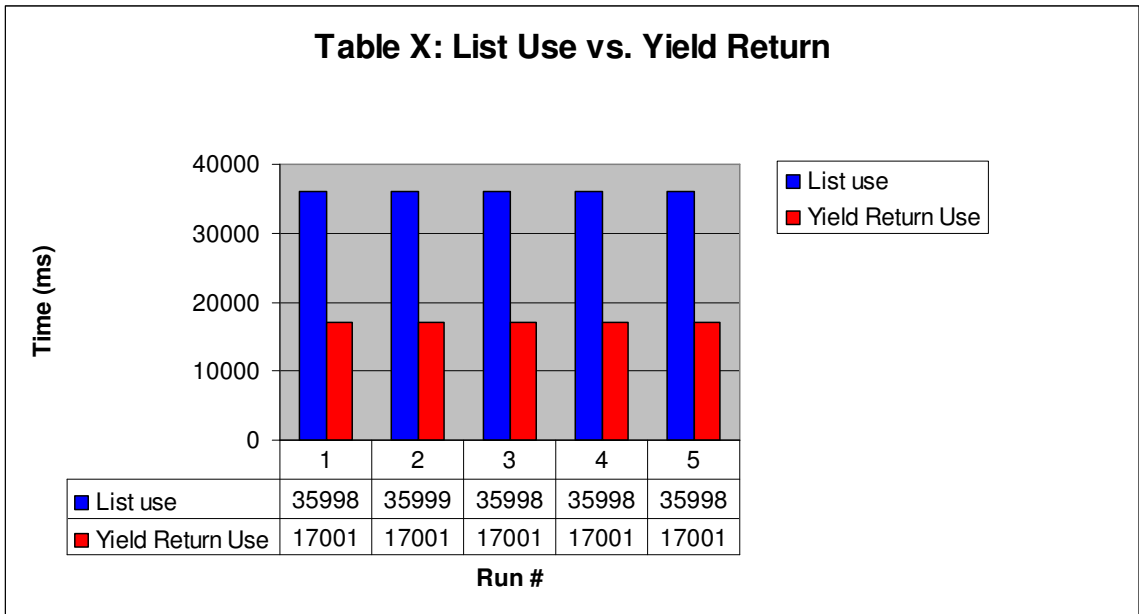
        IEnumerable<double> powers2 = getPowers2(2, 32);
        foreach (double d in powers2)
        {
            if (d == 65536.0) break;
        }
    }

    static List<double> getPowers(int baseNumber, int power)
    {
        List<double> returnList = new List<double>();
        for(int i = 0; i < power; i ++ )
        {
            double d = Math.Pow(baseNumber, i);
            Thread.Sleep(1000);
            returnList.Add(d);
        }
        return returnList;
    }

    static IEnumerable<double> getPowers2(int baseNumber, int power)
    {
        for (int i = 0; i < power; i++)
        {
            double d = Math.Pow(baseNumber, i);
            Thread.Sleep(1000);
            yield return d;
        }
    }
}
```

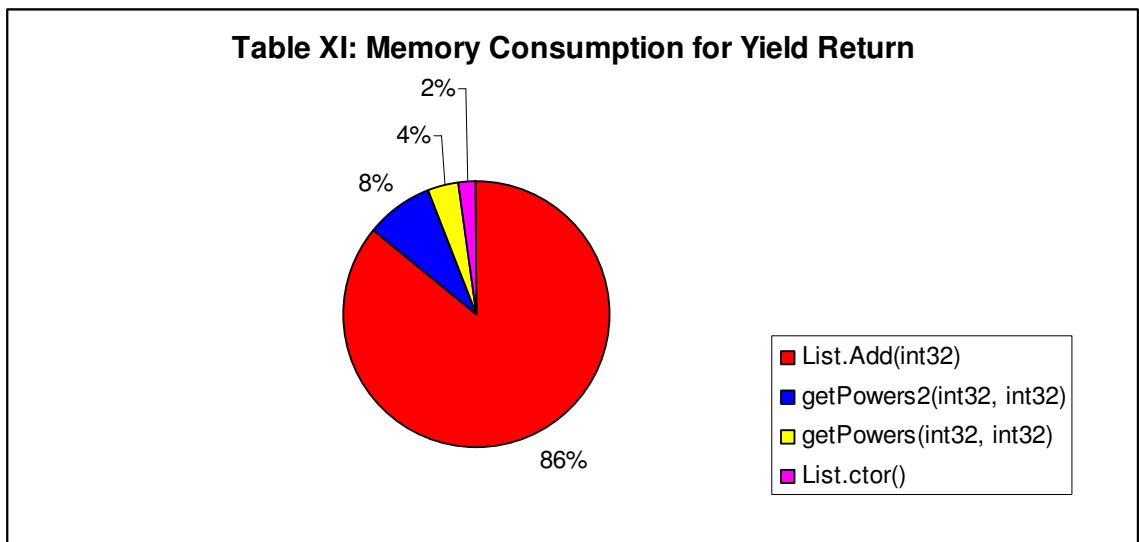
Figure 20: Effective use of yield return.

What yield does is allow us to search on the fly for what we wish to find, instead of having to keep track of the entire list. Table X shows the timings from running this test that works on a list containing all powers of two up to the 32nd degree. We then search for the middle-case (2^{16}) to simulate an average search case.



We see that the timing difference is quite significant; yield return performs over 50% faster, consistently. This average case timing difference will only increase as the lengths of the collections in question grow. In the event that what we do with yield return ends up reaching the end of the list, the timings will be equal. So, this means that yield returning through a list in this matter will never perform worse than a standard list.

Also, upon running some memory profiling built into Visual Studio 2010, we can see that the yield return version is better with regards to memory. In Table XI, we see that using a standard list uses over 85% of the memory that our entire test program uses. The yield return usage counts for a mere 8% of our overall program. This is pocket change for our system. It is clear that the use of yield return allows for the traversal of large data structures with far less impact on the memory of the system. This is a strong point to consider if you are within tight memory constraints.



In an article by Jon Skeet entitled “Iterator block implementation details: auto-generated state machines”, we see a detailed explanation of what transpires behind the scenes with iterators in C#. The main point of focus in this article is the fact that yield return creates a state

machine behind the scenes. This is what allows collections to be rapidly traversed with minimal overhead. The majority of this work is done within the function **MoveNext()**, as stated by Skeet. This function is implemented by the compiler for us when we make use of iterators, and it is not something we as programmers have to actually implement or call. The compiler sets everything up for us. Figure 21 shows the **MoveNext()** implementation generated for our particular use of yield return. This implementation is on par with the examples demonstrated by Skeet in his own article, and serves to explain just how the state machine is being handled. The compiler keeps track of its current state. These states can be the current iterator location, or other states such as -2 which indicates that **GetEnumerator()** has not been called, 0, which is the ready state, and -1, which indicates that the iterator is running, or that it is finished execution.

When stepping through the code line by line with a debugger, some non traditional C# behavior takes place with respect to the **IEnumerable**. This functions as a visual demonstration of what yield return does. We've set an **IEnumerable** instance equal to a function which returns an **IEnumerable**. Our **foreach** loop inspects this **IEnumerable** as

if it were a normal function, and exits out when a yield return is hit. It keeps track of where it left off for the next iteration of the loop. This is the state machine in action (Figure 21).

```
private bool MoveNext()
{
    switch (this.<>1__state)
    {
        case 0:
            this.<>1__state = -1;
            this.<i>5__1 = 0;
            while (this.<i>5__1 < this.power)
            {
                this.<d>5__2 = Math.Pow((double) this.baseNumber, (double) this.<i>5__1);
                Thread.Sleep(0x3e8);
                this.<>2__current = this.<d>5__2;
                this.<>1__state = 1;
                return true;
            }
            Label_006A:
            this.<>1__state = -1;
            this.<i>5__1++;
            break;

        case 1:
            goto Label_006A;
    }
    return false;
}
```

Figure 21: The state machine for Yield Return

Within this state machine created by the compiler, we also see an exciting relic of the past in the form of a **goto** statement. The elusive, never to be used, “spaghetti-code” generating beast from the days of BASIC has now been shown to be beneficial and even required for the use of yield return to function properly. Not that **goto** statements are really that bad in the first place, as they are just the same as your standard jmp (jump to label)

instruction in assembly languages such as 6502, z80, and 8086. This just serves to prove that taboo coding practices of the past have their place in today's programming world, and should not be ignored.

Despite the increased performance we can experience using iterators, we should note that they are also limited in their uses and can even produce dangerous and/or inefficient code almost on accident (much like lambdas). This was hinted at in Skeet's article when he noted that calling **GetEnumerator()** from other threads, or when the current state machine is not in state -2, will result in a new instance being generated to keep track of the new state. This means you now have two counters to deal with. This can keep occurring, and may result in a mess of inconsistency.

These iterators can also produce inefficient operations disguised as extremely concise portions of code as demonstrated in an article entitled "All About Iterators", by Wes Dyer. In the section detailing the cost of iterators, Dyer first examines the **Concat()** sequence operator. This operation contains two **foreach** statements which each contain a **yield return** statement. His test of this statement reveals that the runtime is proportional to the square of the number of **Concat()**s composed together,

or, $O(m^2)$. This was determined by taking **Concat**'s time complexity of $O(m+n)$, where m is the number of items in sequence 1, and n is the number of items in sequence 2. In his example, n is always 1. This means that the calls will then have times ranging from $O((m-1)+1)$ to $O((m-2)+1)$, all the way to $O(1+1)$. Since there are m calls like this, we get $O(m^2)$.

Taking this one step further, Dyer examines recursively defined data structures that are traversed with yield return, specifically, a preorder traversal. This is done with a **foreach** doing the traversal by **yield returning**. This **foreach** is placed within another **foreach** that gets all of the children nodes. So, a nested **foreach** of **yield return** statements. This produces very clean, very concise code (3 simple lines). However, the recursive **yield return** statements cause extra allocations of memory that need not actually be done.

A more appropriate way to take care of this same operation is to use a stack to keep track of what to do as the traversal takes place. It avoids the recursive iterator allocations. As a result, it causes the same operation to perform over twice as quickly, with less memory allocation overall. However, the greatest cost of traversal in either case is still determined by the node

count ($O(b^d)$, where b is the branching factor and d is the depth).

Dyer concludes by referencing a paper written by Bart Jacobs, Frank Piessens, and Wolfram Shulte in which they detail the use of nested iterators to improve their effective performance. This would allow for things such as **Concat ()** and recursive iterator use to perform much better. This is achieved by avoiding redundant evaluation of sequences with **Concat ()** and by keeping track of things with a stack much like Dyer suggested for recursive operations.

They then demonstrate that recursive operations could be done by using a **yield foreach** along with **yield return** statements to recursively operate. This nested iterator style would operate linearly. Currently, a C# iterator would have quadratic performance while achieving the same outcome. This will get very inefficient very quickly with little incentive to actually use it, as the nested iteration pattern would operate far quicker, and with less allocation.

Also noted in "C# In Depth", is that the use of **yield returns** cannot ever guarantee that the iterator will ever be revisited. It is entirely possible that the caller may never return to finish evaluating the rest of a collection. This may not always be a problem, but it should at least be

noted that **yield returns** do not guarantee complete traversal. This also means it is generally poor form to allow for a collection to be modified inconsistently. If a collection is conditionally modified (elements are added or removed), the **yield return** position will suddenly become incorrect. We will not return to where we expect to, and may then create problems for ourselves either by revisiting an element we already processed, or skipping over elements altogether.

Despite these small quirks, our findings agree quite well with both Jon Skeet and Wes Dyer's conclusion that the proper use of iterators allow for very clean code that removes a lot of tedium from the programmer's plate and even serve to make the program itself operate quicker. The quirks of iterators and the use of **yield return** are even difficult to stumble upon without really trying to create a problem, so it is generally a good idea to use them when possible. You gain memory efficiency, cleaner code, and faster traversal results, all through the use of a new feature of C#.

CHAPTER IV

CONCLUDING REMARKS

4.1 Final Verdict

We have covered many topics of C# ranging from anonymous programming, all the way to dynamic programming using reflection and fast iterators.

What we have ultimately discovered is that the original hypothesis claiming that the efficiency of computation is sacrificed in the name of productivity is often false.

The work done behind the scenes by the compiler is in fact surprisingly close to optimal. With respect to anonymous programming, we see that the overhead required to set these functions up is quickly surpassed by the runtime of more important operations within a program. This one time setup cost typically happens faster than a human can even recognize, and the productivity gained from it is very

impactful. Anonymous functions allow for very streamlined, tight code to be created by bypassing normal conventions.

Because of the advances of computing, this anonymous style of programming costs almost nothing in the grand scheme of things and it opens the C# language up to the benefits of clean, event driven programming through the use of delegates. Without anonymous programming methods, event driven programming would not be nearly as clean, or nearly as versatile. We can leave the archaic event driven programming styles behind us and move on to straight forward, flexible event handling.

Even greater than this, we see that the current state of JIT compilation is quite powerful and takes into account the paranoia of seasoned assembly programmers who are now using C#. The end result is very optimal code that is produced on the fly for us. We need not concern ourselves with the specifics of a given platform to gain the most speed because it is done for us. The JIT works in conjunction with the CLR and allows us to target specific platforms and take advantage of the slight nuances of one architecture versus another, all with the same exact C# code.

The current state of JIT compilation within C# is so versatile that we see that the only way for it to truly be

improved would be to introduce some very advanced, specialized design philosophies into the mix such as trace compilation. This kind of JIT compilation strategy would of course benefit more than just the C# JIT compiler. It would affect JIT compilation for any programming language which uses its philosophies in it's' design. Until these JIT optimization strategies are fully realized, we can rest assured that the current status of the JIT compilation strategies used by the CLR are more than adequate to produce very tight, streamlined code that is not susceptible to flaws brought on by code poorly written by the user. It also works so well that we need not try to massage our code in order to gain extra efficiency. It is smart enough to do it for us.

Further proof of the effective use of C# comes from reflection, where the programmer can use seemingly unconventional means to produce dynamic code. The possibilities of reflection are almost endless. For example, the programmer could generate an entire class on the fly and let the compiler take care of all of the details, storing the result dynamically for use later. This stored result eliminates the need to generate it on the fly again and again, which saves us from penalizing

ourselves constantly to take advantage of the dynamic nature of reflection.

While we do experience some overhead and incur a bit of a runtime penalty with reflection, the case is similar to anonymous programming. The performance impact is quickly outweighed by the benefits gained. The dynamic, flexible nature of reflection often proves to win out over the performance hit incurred by letting the compiler take care of the work for us. We saw that the runtime penalty incurred often becomes insignificant, as evidenced by the fact that both reflection and standard **new** operation usage generate a substantial amount of objects per second.

Finally, we looked at the use of iterators with **yield return**. The result was that we can achieve amazing memory performance gains with improved runtime performance as well when we iterate over structures by processing elements one at a time.

We did see that there are some cases where the use of **yield return** could perform poorly, but we also saw how to avoid these pitfalls, along with speculation on future improvements to these iterators. As a whole, the performance gain, and concise code produced by the use of iterators means their use should be strongly encouraged.

The current speculation on improvements to them also shows some promise that they could get even better in the future.

Nearly every downfall with the new features within C# has really come down to simple preference. With anonymous programming, a programmer simply needs to decide if they would like to impact performance slightly and quite possibly insignificantly, while gaining the benefits of streamlined code. They will need to understand that they may have a performance hit at first, but will eventually cross a threshold where the rest of their program's runtime outweighs the anonymous overhead.

With reflection, they again need to decide if they would like to take a performance hit while gaining the benefit of dynamic, flexible code that is cached for future use and may even allow them to achieve things not easily done without reflection.

Every instance of using the wide array of C# features boils down to the programmer weighing the pros and cons of each feature. Fortunately, even if the programmer does not do a good job of weighing the pros and cons, the compiler itself is effective enough that it will manage to optimize out problematic code so that the runtime of a poorly planned out program will not be drastically different than a similar program written more carefully.

What this finally means is that the old style of programming is on its way towards entering a sort of hibernation. We as programmers do not need to concern ourselves as much with the underlying semantics of how the machine operates anymore. We can instead focus on the higher level design of a program to ensure that we get the most out of the language. We do this by utilizing all of the features of the C# language that have been provided to us. It's a new frontier, and we should tread into it confidently, using these new tools to our utmost advantage.

BIBLIOGRAPHY

Bouakaz, Adnan, Isabelle Puaut, and Erven Rohou.

"Predictable Binary Code Cache: A First Step Towards Reconciling Predictability and Just-In-Time Compilation." Web.

Lowy, Juval. "C#: Coding With Anonymous Methods, Iterators, And Partial Classes." Web.

<<http://msdn.microsoft.com/en-us/magazine/cc163682.aspx>>.

Dyer, Wes. "Yet Another Language Geek." *All About Iterators*. Web.

<<http://blogs.msdn.com/b/wesdyer/archive/2007/03/23/all-about-iterators.aspx>>.

Gal, Andreas. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. Thesis. University of California, Irvine, 2006. Print.

Goldshtein, Sasha. "JIT Optimizations." - *CodeProject*. Web.

<<http://www.codeproject.com/Articles/25801/JIT-Optimizations>>.

Grunwald, Daniel. "ILSpy - Yield Return." *SharpDevelop Community*. Web.

<<http://community.sharpdevelop.net/blogs/danielgrunwald/archive/2011/03/06/ilspy-yield-return.aspx>>.

Guo, Shu-yu, and Jens Palsberg. "The Essence of Compiling with Traces." Web.

McMullen, Eric. "Get Drunk on the Power of Reflection.Emit." *DevX*. Web.
<<http://www.devx.com/dotnet/Article/28783/1954>>.

Meier, JD, Srinath Vasireddy, Ashish Babbar, Rico Mariani, and Alex Mackman. "Chapter 5 - Improving Managed Code Performance." Web.
<<http://msdn.microsoft.com/en-us/library/ff647790.aspx>>.

Nash, Trey. *Accelerated C# 2008*. Berkeley, CA: Apress, 2007. Print.

Richter, Jeffrey. *CLR via C#*. Redmond, WA: Microsoft, 2006. Print.

Richter, Jeffrey. ".NET: An Introduction to Delegates." Web.
<<http://msdn.microsoft.com/en-us/magazine/cc301810.aspx>>.

Robinson, Simon. *Advanced .NET Programming*. Birmingham: Wrox, 2002. Print.

Saravanakumar, Aarthi. ".NET Framework - Internals Of Delegate Chaining." *EggHeadCafe*. Web.
<http://www.eggheadcafe.com/articles/dotnet_delegates.asp>.

Skeet, Jon. "Iterator Block Implementation Details: Auto-generated State Machines." *C# in Depth: Iterator Block Implementation Details*. Web.
<<http://csharpindepth.com/Articles/Chapter6/IteratorBlockImplementation.aspx>>.

Wagner, Bill. *Effective C#: 50 Specific Ways to Improve Your C#*. Upper Saddle River, NJ: Addison-Wesley, 2010. Print.

APPENDIX

APPENDIX A
(Full Class Listings and Disassemblies)

```
public class delegateClass
{
    public delegateClass() { }

    public int add(int x, int y)
    {
        return x + y;
    }
    public int sub(int x, int y)
    {
        return x - y;
    }
    public int div(int x, int y)
    {
        return x / y;
    }
    public int mul(int x, int y)
    {
        return x * y;
    }
    public int pow(int x, int y)
    {
        int r = 1;
        for (int i = 0; i < y; i++)
        {
            r *= x;
        }
        return r;
    }
}
```

Full delegateClass class listing.

```

/////PROCESS #1/////
IL_0015: ldnull
IL_0016: stloc.1
IL_0017: ldnull
IL_0018: stloc.3
IL_0019: newobj      instance void tests/Program/'<>c__DisplayClass3'::.ctor()
IL_001e: stloc.s     'CS$<>8_locals4'
IL_0020: ldloc.s     'CS$<>8_locals4'
IL_0022: ldc.i4.1
IL_0023: stfld      int32 tests/Program/'<>c__DisplayClass3'::i
IL_0028: br.s        IL_0059
IL_002a: nop
IL_002b: ldloc.3
IL_002c: brtrue.s   IL_003e
IL_002e: ldloc.s   'CS$<>8_locals4'
IL_0030: ldftn      instance void tests/Program/'<>c__DisplayClass3'::'<Main>b_0'()
IL_0036: newobj      instance void tests/Program/MyDelegate::.ctor(object, native int)
IL_003b: stloc.3
IL_003c: br.s        IL_003e
IL_003e: ldloc.3
IL_003f: stloc.2
IL_0040: ldloc.2
IL_0041: stloc.1
IL_0042: ldloc.1
IL_0043: callvirt   instance void tests/Program/MyDelegate::Invoke()
IL_0048: nop
IL_0049: nop
IL_004a: ldloc.s   'CS$<>8_locals4'
IL_004c: dup
IL_004d: ldfld      int32 tests/Program/'<>c__DisplayClass3'::i
IL_0052: ldc.i4.1
IL_0053: add
IL_0054: stfld      int32 tests/Program/'<>c__DisplayClass3'::i
IL_0059: ldloc.s     'CS$<>8_locals4'
IL_005b: ldfld      int32 tests/Program/'<>c__DisplayClass3'::i
IL_0060: ldc.i4.5
IL_0061: cgt
IL_0063: ldc.i4.0
IL_0064: ceq
IL_0066: stloc.s   CS$4$0000
IL_0068: ldloc.s   CS$4$0000

```

Figure 8: MSIL Disassembly


```

////////////////////////////////////
IL_0097: ldnull
IL_0098: stloc.s    d2
IL_009a: ldc.i4.1
IL_009b: stloc.s    i
IL_009d: br.s      IL_00d6
IL_009f: newobj     instance void tests/Program/'<>c__DisplayClass5'::.ctor()
IL_00a4: stloc.s    'CS$<>8_locals6'
IL_00a6: nop
IL_00a7: ldloc.s    'CS$<>8_locals6'
IL_00a9: ldloc.s    i
IL_00ab: stfld     int32 tests/Program/'<>c__DisplayClass5'::k
IL_00b0: ldloc.s    'CS$<>8_locals6'
IL_00b2: ldftn     instance void tests/Program/'<>c__DisplayClass5'::'<Main>b_1'()
IL_00b8: newobj     instance void tests/Program/MyDelegate2::.ctor(object, native int)
IL_00bd: stloc.s    V_7
IL_00bf: ldloc.s    d2
IL_00c1: ldloc.s    V_7
IL_00c3: call      class [mscorlib]System.Delegate
            [mscorlib]System.Delegate::Combine(class [mscorlib]System.Delegate,
            class [mscorlib]System.Delegate)

IL_00c8: castclass tests/Program/MyDelegate2
IL_00cd: stloc.s    d2
IL_00cf: nop
IL_00d0: ldloc.s    i
IL_00d2: ldc.i4.1
IL_00d3: add
IL_00d4: stloc.s    i
IL_00d6: ldloc.s    i
IL_00d8: ldc.i4.5
IL_00d9: cgt
IL_00db: ldc.i4.0
IL_00dc: ceq
IL_00de: stloc.s    CS$4$0000
IL_00e0: ldloc.s    CS$4$0000
IL_00e2: brtrue.s  IL_009f
IL_00e4: ldloc.s    d2
IL_00e6: callvirt  instance void tests/Program/MyDelegate2::Invoke()
IL_0117: ldc.i4.1
IL_0118: stloc.s    i
IL_011a: br.s      IL_012c
IL_011c: nop

```

Figure 8 (continued): MSIL Disassembly.

```

/////PROCESS # 3 ////////////////////////////////////////
IL_011d: ldloc.s    i
IL_011f: call      void tests/Program::doABunchOfStuff(int32)
IL_0124: nop
IL_0125: nop
IL_0126: ldloc.s    i
IL_0128: ldc.i4.1
IL_0129: add
IL_012a: stloc.s    i
IL_012c: ldloc.s    i
IL_012e: ldc.i4.5
IL_012f: cgt
IL_0131: ldc.i4.0
IL_0132: ceq
IL_0134: stloc.s    CS$4$0000
IL_0136: ldloc.s    CS$4$0000
IL_0138: brtrue.s  IL_011c
IL_0157: ret
} // end of method Program::Main

```

Figure 8 (continued): MSIL Disassembly.

```

IL_0189: ldsfld      class [mscorlib]System.Func`2<class tests/Program/Book,bool>
           tests/Program::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_018e: brtrue.s     IL_01a3
IL_0190: ldnull
IL_0191: ldftn      bool tests/Program::'<Main>b 0'(class tests/Program/Book)
IL_0197: newobj     instance void class [mscorlib]System.Func`2<class tests/Program/Book,bool>
           ::ctor(object, native int)
IL_019c: stsfld     class [mscorlib]System.Func`2<class tests/Program/Book,bool> tests/Program
           ::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_01a1: br.s       IL_01a3
IL_01a3: ldsfld     class [mscorlib]System.Func`2<class tests/Program/Book,bool> tests/Program
           ::'CS$<>9__CachedAnonymousMethodDelegate2'
IL_01a8: call      class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>
[System.Core]System.Linq.Enumerable::Where<class tests/Program/Book>
           (class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>, class
           [mscorlib]System.Func`2<!!0,bool>)

IL_01ad: stloc.s   scifi
IL_01bb: ldloc.0
IL_01bc: callvirt   instance valuetype [mscorlib]System.TimeSpan
           [System]System.Diagnostics.Stopwatch::get_Elapsed()
           [mscorlib]System.TimeSpan
IL_01c1: box
IL_01c6: ldloc.s   scifi
IL_01c8: call      int32 [System.Core]System.Linq.Enumerable::Count<class tests/Program/Book>
           (class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>)
           [mscorlib]System.Int32
IL_01cd: box
IL_01d2: call      void [mscorlib]System.Console::WriteLine(string,object, object)
IL_01e6: newobj     instance void class [mscorlib]System.Collections.Generic.List`1
           <class tests/Program/Book>::ctor()

IL_01eb: stloc.s   scifi2
IL_01ed: nop
IL_01ee: ldloc.1
IL_01ef: callvirt instance valuetype [mscorlib]System.Collections.Generic.List`1/Enumerator<!!0>
class [mscorlib]System.Collections.Generic.List`1<class tests/Program/Book>::GetEnumerator()
IL_01f4: stloc.s   CS$5$0001
.try {
  IL_01f6: br.s     IL_0227
  IL_01f8: ldloc.s   CS$5$0001
  IL_01fa: call      instance !0 valuetype [mscorlib]System.Collections.Generic.List`1
           /Enumerator<class tests/Program/Book>::get_Current()

  IL_01ff: stloc.s   b
  IL_0201: nop
  IL_0202: ldloc.s   b
  IL_0204: ldfld     string tests/Program/Book::genre
  IL_0209: ldstr     "Science Fiction"
  IL_020e: call      bool [mscorlib]System.String::op_Equality(string, string)
  IL_0213: ldc.i4.0
  IL_0214: ceq
  IL_0216: stloc.s   CS$4$0000
  IL_0218: ldloc.s   CS$4$0000
  IL_021a: brtrue.s   IL_0226
  IL_021c: ldloc.s   scifi2
  IL_021e: ldloc.s   b
  IL_0220: callvirt   instance void class [mscorlib]System.Collections.Generic.List`1
           <class tests/Program/Book>::Add(!0)

  IL_0225: nop
  IL_0226: nop
  IL_0227: ldloc.s   CS$5$0001
  IL_0229: call      instance bool valuetype [mscorlib]System.Collections.Generic.List`1
           /Enumerator<class tests/Program/Book>::MoveNext()

  IL_022e: stloc.s   CS$4$0000
  IL_0230: ldloc.s   CS$4$0000
  IL_0232: brtrue.s   IL_01f8
  IL_0234: leave.s   IL_0245
} // end .try

```

Figure 12: MSIL for searching & sorting with lambdas and procedural code

```

finally
{
  IL_0236: ldloc.s CS$5$0001
  IL_0238: constrained. valuetype [mscorlib]System.Collections.Generic.List
          `1/Enumerator<class tests/Program/Book>
  IL_023e: callvirt instance void [mscorlib]System.IDisposable::Dispose()
  IL_0243: nop
  IL_0244: endfinally
}
IL_0253: callvirt instance valuetype [mscorlib]System.TimeSpan
          [System]System.Diagnostics.Stopwatch::get_Elapsed()
IL_0258: box [mscorlib]System.TimeSpan
IL_025d: ldloc.s scifi2
IL_025f: callvirt instance int32 class [mscorlib]System.Collections.Generic.List`1
          <class tests/Program/Book>::get_Count()
IL_0264: box [mscorlib]System.Int32
IL_0269: call void [mscorlib]System.Console::WriteLine(string,object, object)
IL_027d: ldloc.s scifi
IL_027f: ldsfld class [mscorlib]System.Func`2<class tests/Program/Book,string>
          tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_0284: brtrue.s IL_0299
IL_0286: ldnull
IL_0287: ldftn string tests/Program::'<Main>b_1'(class tests/Program/Book)
IL_028d: newobj instance void class [mscorlib]System.Func`2<class
          tests/Program/Book,string>::ctor(object,
          native int)
IL_0292: stsfld class [mscorlib]System.Func`2<class tests/Program/Book,string>
          tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_0297: br.s IL_0299
IL_0299: ldsfld class [mscorlib]System.Func`2<class tests/Program/Book,string>
          tests/Program::'CS$<>9__CachedAnonymousMethodDelegate3'
IL_029e: call class [System.Core]System.Linq.IOrderedEnumerable`1<!!0>
          [System.Core]System.Linq.Enumerable::OrderBy<class
          tests/Program/Book,string>(class [mscorlib]System.Collections.
          class [mscorlib]System.Func`2<!!0,!!1>)
IL_02a3: stloc.s byName
IL_02a5: ldloc.0
IL_02a6: callvirt instance void [System]System.Diagnostics.Stopwatch::Stop()
IL_02ab: nop
IL_02ac: ldstr "Lambda sort took {0}."
IL_02b1: ldloc.0
IL_02b2: callvirt instance valuetype [mscorlib]System.TimeSpan
          [System]System.Diagnostics.Stopwatch::get_Elapsed()
IL_02b7: box [mscorlib]System.TimeSpan
IL_02bc: call void [mscorlib]System.Console::WriteLine(string, object)
IL_02d0: ldloc.s scifi2
IL_02d2: callvirt instance void class [mscorlib]System.Collections.Generic.List`1
          <class tests/Program/Book>::Sort()
IL_02e5: callvirt instance valuetype [mscorlib]System.TimeSpan
          [System]System.Diagnostics.Stopwatch::get_Elapsed()
IL_02ea: box [mscorlib]System.TimeSpan
IL_02ef: call void [mscorlib]System.Console::WriteLine(string,object)
IL_02f4: nop
IL_02f5: ret
} // end of method Program::Main

```

Figure 12 (continued): MSIL for searching & sorting with lambdas and procedural code