An Investigation of String and String Builder Classes in C#
Daniela Gotseva
Assoc. Prof. PhD, Department of Computer Systems, Technical University of Sofia, Bulgaria
dgocova@tu-sofia.bg

ABSTRACT
The performance of string and String Builder is shown in this paper. This research is applicable to C# language only. After short explanation about string and String Builder with their basic properties, the tests are provided. For test purposes Visual Studio 2012 Express Edition is used. The results are shown in charts and some proposals for using of these data types are given. At the end of the paper some suggestion about how to increase String Builder performance are made.

Keywords: String, String Builder, C#

1. INTRODUCTION
A string [1, 3] is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters (‘0’). The Length property of a string represents the number of Char objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, we use the String Info object. String objects are immutable: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object.

A String Builder [2-3] represents a mutable string of characters. This class cannot be inherited. It can be changed as many times as necessary. This yields big performance improvements. It eliminates many string copies and in certain loops, which is essential.

Strings are so heavily used in all programming languages that we do not think about them very much. We use them simply and hope to do the right thing. Normally all goes well but sometimes we need more performance so we switch to String Builder which is more efficient because it does contain a mutable string buffer. .NET Strings are immutable which is the reason why a new string object is created every time we alter it (insert, append, remove, etc.).

That sounds reasonable, so why do we still use the .NET String class functions and not the faster String Builder? Because optimal performance is a tricky thing and the first rule of the performance club is to measure it. It is very difficult to predict the performance of some code in advance because we have to know so many variables that influence the outcome. Looking at the generated MSIL code does still NOT tell us how fast the code will perform. If you want to see why our function is so slow/fast we have to look at the compiled (JIT ed) x86 assembler code to get the full picture.

2. TEST CASES
In the following article we will show you the numbers for String Builder vs String which we did measure with .NET 4.5 a Intel Core i7 2.2 GHz with 16.0 GB RAM.

Every test was performed 10 million times to get a stable value.

The test cases are created to give answer to the followed questions:

- What data type is better to use in insert, remove, and replace operations? The answer is shown in test T1.
- What is the performance of string, Format s() and String Builder. Append Format () methods? The decision of this question is in test T2.
- What method for string concatenation is the best? The result is present in test T3.

For testing purpose the C# program is created, according to [3, 4]. For each test case special function is created and called via delegate. The test functions are shown in each test case.

2.1 Test case T1: String and String Builder Operations
We inserted the some words (1, 5, 10, and 15) at the beginning of the arbitrary sentence to find out the breakeven point between String, Insert and String Builder. Insert. The test functions are present in Listing 1. The results are shown in figure 1.
To see how the removal of characters worked we removed in a for loop one character from the beginning of our test sentence. The results are shown in the figure 2. The test functions are present in Listing 2.

```
Listing 1: Insert Operation Test Code

string StringInsert(string str, string[] inserts)
{
    foreach (string insert in inserts)
    {
        str = str.Insert(0, insert);
    }
    return str;
}

string StringBuilderInsert(string str, string[] inserts)
{
    StringBuilder sb = new StringBuilder(str);
    foreach (string insert in inserts)
    {
        sb.Insert(0, insert);
    }
    return sb.ToString();
}
```

```
Listing 2: Remove Operation Test Code

string StringRemove(string str, int Count)
{
    for (int i = 0; i < Count; i++)
    {
        str = str.Remove(0, 1);
    }
    return str;
}

string StringBuilderRemove(string str, int Count)
{
    StringBuilder sb = new StringBuilder(str);
    for (int i = 0; i < Count; i++)
    { sb.Remove(0, 1); }
    return sb.ToString();
}
```

![Fig 1: String. Insert vs StringBuilder.Insert](image1)

![Fig 2: String. Remove vs StringBuilder.Remove](image2)

We see here that String Builder is clearly the better choice if we have to alter the string. Insert and Remove operations are nearly always faster with String Builder. The removal of characters is especially fast with String Builder where we gain nearly a factor of two.

Things do become more interesting when we do replace anywhere from one to five words of our test sentence. The results is shown in figure 3. The test functions are present in Listing 2. Here we used List collection with 2 fields: old text that should be replaced by new string.
This is somewhat surprising. String Builder does not beat String. Replace even if we do many replaces. There seems to be a constant overhead of about 1 sec we see in our data that we pay if we use String Builder. The overhead is quite significant (30%) when we have only a few String Replaces to do.

2.2 Test case T2: String and String Builder Format

Here we checked when String Builder. Append Format is better than String. Format. The code sample is shown in Listing 4, and the result chart is present in figure 4.

2.3 Test case T3: String and String Builder Concatenations

This is the most interesting test because we have several options here. We can concatenate strings with +, String. Concat, String. Join and String Builder. Append.

The winner is String. Join. After taking a deep look with Reflector we found that String. Join has the most efficient algorithm implemented which allocates in the first pass the final buffer size and then memcopy each string into the just allocated buffer. This is simply unbeatable. String Builder does become better above 7 strings compared to the + operator but this is not really code one would see very often.

The code listing is shown in Listing 5. The result chart is published in figure 5.
3. STRINGBUILDER PERFORMANCE

String Builder is mainly a performance optimization for the string type in certain cases. In this section, we will demonstrate the performance and memory usage of String Builder.

3.1 Short Appends

String Builder optimizes string appends. However, it does not eliminate the cost of individual appends. This benchmark shows that performance improves by appending fewer, larger strings. To test this case a new approach is created. The goal is to create a String Builder with five million characters in it. we append individual strings of lengths 1 to 99 characters to reach that final length. The Append method calls are timed. The result is shown in Figure 6.

Programs that use String Builder will perform faster if some calls to Append are combined if possible. Coalescing Append calls could lead to a performance gain. However, this only helps when appending strings less than about 20 characters.

3.2 Cache Objects

String Builder is an object. It is used as an optimization. In many programs the String Builder itself is created many times. With a String Builder cache you can reduce this cost further. We present an example program with two methods. The first version, Method1, creates a new String Builder in every call. It then adds strings to the String Builder and returns the string representation. The second version, Method2, references a static String Builder field and then clears it each time. The String Builder field will remain allocated in one part of memory throughout all calls to Method2. The result is shown in Listing 6.

In conclusion, if we adjust the capacity for the String Builder, Method1 will perform faster than before but still slower than Method2. This is because Method2 does not require capacity adjustments after the first call in the benchmark.

Using the String Builder type as a field can reduce memory allocations and improve performance. Unfortunately, you do need to call Clear every time it is accessed or the results will be incorrect.
3.3 Data Types

String Builder appends different data types with overloads. What data types are fastest to Append? We can optimize String Builder code by choosing between data types when possible. When you call the Append or Append Line methods on String Builder, you can pass different types. There are 19 overloads of these functions.

The benchmark is divided into three pairs. The first pair tests a one-character int against the same text in char format. The second pair tests a bool against the same text in string format. The third pair tests a char against its same text in string format. It uses a one-character string.

We compared equivalent String Builder appended values. Note that the code samples may not be exactly equivalent in all cases. It is usually fastest to append chars, then strings, and then other types. Ints are the slowest here. Here is the result of type’s performance, which is shown in Listing 7.

3.4 Prefer Chars

It is possible to optimize the String Builder Append method. This applies to certain cases that deal with short strings. Instead of appending a string literal that contains two characters, we append two characters separately.

This program benchmark demonstrates how you can express a two-characters append operation in two different ways. In the first tight loop, two space characters are appended in separate method calls. In the second tight loop, a two-character string is appended in a single method call. The first loop has better performance. The result is shown in Listing 8.

3.5 Prefer Strings

String Builder is an optimization. But it doesn't always improve performance. We determine the point at which String Builder becomes faster. Here we see benchmarks of String Builder and strings. We discuss the important considerations.

At some point, using separate character appends will slow down the method you are writing. We found using chars for 1, 2, 3 and often 4 character long strings was more efficient. The strings were more efficient longer than that.
inclusive. Using StringBuilder with an accurate capacity is always faster than without.

3.6 Append Digits

Appending an integer as a string is inefficient. This is the case with StringBuilder or Http Response in the .NET Framework. The implementation must convert the integer to a string. This is allocated upon the managed heap.

Let's look at the benchmark harness itself, which tests the Append method on the StringBuilder type to the custom Digits method. Each StringBuilder is appended to with the numbers 0 - 19999. The program uses the Digits extension method. This method receives an integer parameter. It extracts each digit from the integer at a time from left to right, and then appends it to the underlying buffer. The result from test is shown in Listing 10. As we can see, the Digits method performed the task nearly 1.5 milliseconds faster.

3.7 Set Capacity

StringBuilder in .NET allocates its capacity. When it runs out of space to Append characters, it reallocates and copies the data, leading to slowdowns. It uses a clever algorithm for this.

For testing purpose the console application in C# is created. It loops through 257 integers, 0 to 256 inclusive. It writes the Capacity and Length of the StringBuilder, and then appends one character. The code is present in Listing 11. The results are shown in Figure 8.

Listing 9: The Three Testing Methods

```csharp
    static string StringBuilderDefault(string s, int count1)
    {
        StringBuilder builder1 = new StringBuilder();
        for (int i = 0; i < count1; i++)
        {
            builder1.Append(s);
        }
        return builder1.ToString();
    }
    static string StringBuilderWithCapacity(string s, int count1, int maxlength)
    {
        StringBuilder builder2 = new StringBuilder(count1 * maxlength);
        for (int i = 0; i < count1; i++)
        {
            builder2.Append(s);
        }
        return builder2.ToString();
    }
    static string StringConcat(string s, int count1)
    {
        string st1 = string.Empty;
        for (int i = 0; i < count1; i++)
        {
            st1 += s;
        }
        return st1;
    }
```

Listing 10: Append Digit Benchmark Program

```csharp
    const int _max = 10000;
    GC.Collect();
    StringBuilder b1 = new StringBuilder();
    var s1 = Stopwatch.StartNew();
    for (int i = 0; i < _max; i++)
    {
        b1.Append(a);
    }
    s1.Stop();
    GC.Collect();
    StringBuilder b2 = new StringBuilder();
    var s2 = Stopwatch.StartNew();
    for (int i = 0; i < _max; i++)
    {
        b2.Digits(a);
    }
    s2.Stop();

    Output:
    1.94 ms
    0.56 ms
```

Fig 7: Results from Concatenation Benchmark

3.6 Append Digits

Append an integer as a string is inefficient. This is the case with StringBuilder or Http Response in the .NET Framework. The implementation must convert the integer to a string. This is allocated upon the managed heap.

For testing purpose the console application in C# is created. It loops through 257 integers, 0 to 256 inclusive. It writes the Capacity and Length of the StringBuilder, and then appends one character. The code is present in Listing 11. The results are shown in Figure 8.
4. CONCLUSION

The following recommendations are valid for our small test strings (~30 chars) but should be applicable to bigger strings (100-500) as well (measure for yourself!). We have seen many synthetic performance measurements that demonstrate the power of String Builder with strings that are 10KB and bigger. This is the 1% case in real world programs. Most strings will be significantly shorter. When you optimize a function and you can “feel” the construction costs of an additional object then you have to look very carefully if you can afford the additional initialization costs of String Builder. The summarized results are shown in Table 1.

<table>
<thead>
<tr>
<th>String Operation</th>
<th>Most Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>StringBuilder.Insert for greater than 2 Insertion Strings, otherwise String. Insert</td>
</tr>
<tr>
<td>Remove</td>
<td>String Builder is faster for greater than 2 characters to remove</td>
</tr>
<tr>
<td>Replace</td>
<td>String. Replace always</td>
</tr>
<tr>
<td>Format</td>
<td>String. Format for less than 5 Append + Format operations. otherwise StringBuilder.AppendFormat</td>
</tr>
<tr>
<td>Concatenation</td>
<td>+ for 2 strings, otherwise String. Join</td>
</tr>
</tbody>
</table>

REFERENCES

AUTHOR PROFILES
D. Gotseva received the degree in computer science from Technical University of Sofia, in 2005. Currently, she is an Associate Professor at Computer Systems Department in Faculty of Computer Systems and Control.