# Highlighted Performance Wins with .NET 7

Ever since .NET Core hit the scene more than seven years ago, performance has been an integral part of the culture of .NET. Developers are encouraged to find and improve all aspects of performance across the codebase, with changes ranging from removing a small allocation here, to overhauling an entire algorithm there, to brand new APIs intended to enable all developers to further improve their own application and service performance. This, in turn, leads to immense quantities of improvements in any given release, and .NET 7 is no exception. The post at <https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7> provides an in-depth exploration of hundreds of these.

Given the quantity and quality of these improvements, there’s no handful of changes I can point to that are “the best,” with changes each having their own unique impact in distinct areas of the platform. Instead, in this article, I’ve picked just three areas of improvement to highlight as ones that can have an instrumental impact on the performance of your code. This is a small taste, and I encourage you to read the (long) cited post to get a better understanding for the breadth and depth of performance improvements in this release.

## On-Stack Replacement

There are many positives to just-in-time (JIT) compilation. A single binary can be run on any supported platform, with the code in the binary translated to the target computer’s architecture on-demand, and in doing so, the JIT can specialize the generated code for the particulars of the target computer, such as choosing the best instructions it supports to implement a particular operation. The flip side of this is that on-demand compilation takes time during the execution of the program, and that time often shows up as delays in the startup of an application (or, for example, in the time for a service to complete a first request).

Tiered compilation was introduced in .NET Core 3.0 as a compromise between startup and steady-state throughput. With it, the JIT is able to compile methods multiple times. It first does so with minimal optimization, in order to minimize how long it takes to compile the method (because the cost of finding and applying optimizations is a significant percentage of the cost of compiling a method). It also equips that minimally optimized code with tracking for how many times the method is invoked. Once the method has been invoked enough times, the JIT recompiles the method, this time with all possible optimization, and redirects all future invocations of the method to that heavily optimized version.

That way, startup is faster while steady-state throughput remains efficient; in fact, steady-state throughput can even be faster because of additional information the JIT can learn about the method from its first compilation and then use when doing the subsequent compilation. However, methods with by-default loops previously were opted-out of tiered compilation because such methods can consume significant amounts of the app’s execution time even without being invoked multiple times.

In .NET 7, even methods with loops benefit from tiered compilation. This is achieved via on-stack replacement (OSR). OSR results in the JIT not only equipping that initial compilation for number of invocations, but also equipping loops for the number of iterations processed. When the number of iterations exceeds a predetermined limit, just as with invocation count, the JIT compiles a new optimized version of the method. It then jumps to the new method, transferring over all relevant state so that execution can continue running seamlessly without the method needing to be invoked again.

You can see this in action in a few ways. Try compiling a simple console app:

using System;

for (int i = 0; ; i++)

{

 if (i == 0)

 {

 Console.WriteLine("Running...");

 }

}

Run it, but first set the **DOTNET\_JitDisasmSummary** environment variable to 1. You should end up seeing output that includes lines like these:

 4: JIT compiled Program:<Main>$(ref)

 [Tier0, IL size=21, code size=94]

…

 6: JIT compiled Program:<Main>$(ref)

 [Tier1-OSR @0x2, IL size=21, code size=43]

The C# compiler generated a **<Main>$** method for your program containing top-level statements, and you see two entries for it in the JIT’s recording of every method it compiled. The first is the initial compilation with minimal optimization, and the second is the OSR variant that’s fully optimized. If you didn’t have OSR such that tiered compilation didn’t apply to this method, you would have instead seen a single entry for this method.

Or, try running this silly little benchmark:

using BenchmarkDotNet.Attributes;

using BenchmarkDotNet.Running;

using System;

[DisassemblyDiagnoser]

public partial class Program

{

 static void Main(string[] args) =>

 BenchmarkSwitcher.

 FromAssembly(typeof(Program).Assembly).

 Run(args);

 private static readonly int s\_year =

 DateTime.UtcNow.Year;

 [Benchmark]

 public int Compute()

 {

 int result = 0;

 for (int i = 0; i < 1\_000\_000; i++)

 {

 result += i;
 if (s\_year == 2021)

 {

 result += i;

 }

 }

 return result;

 }

}

When I run it, I get numbers like .NET 6 taking 858.9us and .NET 7 taking 237.3us. Why is .NET 7 so much faster? Because of OSR.

In the .NET 6 version, tiered compilation isn’t used and the loop is compiled such that the comparison in the body of the loop requires reading the value of **s\_year**. In the .NET 7 version, tiered compilation is used, and when OSR kicks in and causes an optimized version of the code to be generated, by that point, the JIT knows that **s\_year** was already initialized. As it’s a **static readonly**, the JIT knows its value will never change, and the JIT can treat it like a constant. It’ll see that the year isn’t 2021 and so eliminate that **if** block as dead code. You’re thus saving a static field read and a branch on every iteration of the loop when compared to the .NET 6 execution.

## Regex

.NET 7 significantly improves the performance of regular expressions processing, so much so that in addition to the previously cited .NET 7 performance post, there’s an entire post dedicated to **Regex** improvements (<https://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7>). The performance improvements here broadly fit into four categories: ones that result in existing regexes being faster, new APIs for more efficiently working with regexes, a new regex source generator (which has not only startup and steady-state performance benefits but also pedagogical benefits), and the new **RegexOptions.NonBacktracking** option. Here, I’ll take a look at the source generator, touching on a few of the other improvements, and you can see the blog posts for more details.

Let’s say I wanted a regular expression for a specific formatting of phone numbers in the United States:

@"[0-9]{3}-[0-9]{3}-[0-9]{4}"

This looks for three digits, a dash, three more digits, another dash, and another four digits. And let’s say my task was to count the number of phone numbers in a given piece of text. With .NET 6 and earlier, I might implement that as follows:

partial class Helpers

{

 private static readonly Regex s\_pn =

 new Regex(@"[0-9]{3}-[0-9]{3}-[0-9]{4}",

 RegexOptions.Compiled);

 public static int CountPhoneNumbers(

 string text)

 {

 int count = 0;

 Match m = s\_pn.Match(text);

 while (m.Success)

 {

 count++;

 m = m.NextMatch();

 }

 return count;

 }

}

This uses the **RegexOptions.Compiled** flag to ask the runtime to use reflection emit to generate a customized implementation of this regular expression at execution time. Doing so will make steady-state throughput much faster, at the expense of having to do all of the work to parse, analyze, optimize, and code gen this expression at execution time. It also won’t be able to do that code generation in an environment that lacks a JIT compiler, like with Native AOT in .NET 7. What if all of that work could be done at compile-time instead? In .NET 7, it can. I can write the above in .NET 7 instead as the following:

partial class Helpers

{

 [GeneratedRegex(

 @"[0-9]{3}-[0-9]{3}-[0-9]{4}")]

 private static partial Regex PhoneNumber();

 public static int CountPhoneNumbers(

 ReadOnlySpan<char> text) =>

 PhoneNumber().Count(text);

}

A few things to notice here. First, my **CountPhoneNumbers** helper now accepts a **ReadOnlySpan<char>** instead of a **string** (which is implicitly convertible to **ReadOnlySpan<char>**). That’s feasible because **Regex** now has multiple methods that accept **ReadOnlySpan<char>** as input and work efficiently over such spans, enabling my **CountPhoneNumbers** function to be used with a wider set of inputs, such as a **char[]**, a stackalloc’d **Span<char>**, a **ReadOnlySpan<char>** created around some native memory from interop, and so on.

Second, the body of my **CountPhoneNumbers** method has been condensed to a single line that just calls the **Count** method; this is a new method on **Regex** in .NET 7 that efficiently counts the number of occurrences in the input, in an amortized allocation-free manner.

Third and most importantly for this example, you’ll notice that I’m no longer using the **Regex** constructor. Instead, I have a partial method that returns **Regex** and that’s attributed with **[GeneratedRegex(…)]**. This new attribute triggers a source generator included in the .NET SDK to emit a C# implementation of the specified regex, as shown in the screenshot of Visual Studio in **Figure 1**. The generated code is logically equivalent to the IL that would be emitted by **RegexOptions.Compiled** at execution time but is instead C# emitted at compile time. This means that you get all the throughput benefits of the compiled approach, even in an environment that doesn’t support JIT’ing, and you get those benefits without having to pay for it in startup time. This code is also viewable and debuggable, which helps improve not only the correctness of an application but also your knowledge of how regexes are evaluated:



**Figure 1:** Stepping through C# code generated for a Regex

You can see in this example that the source generator has output very clean, very readable source code for the regular expression, tailored to exactly what the regex is, and very close to what you yourself might write if you were trying to write code to match this pattern. The code is even commented, and it uses new methods also introduced in .NET 7 for identifying categories of characters like ASCII digits.

It’s also learned some new tricks in how to search efficiently for possible locations that might match the pattern. In .NET 6, it would have walked character by character looking for a digit that could start the pattern. Now in .NET 7, you can see it’s searching for the dash**,** and if found, it’ll be able to back up a few characters to try to match the pattern starting at that position. In a text like “Pride and Prejudice” that I’m searching here, which, as you might guess, has very few U.S. phone numbers in it, this leads to significant performance improvements.

## LINQ

Language Integrated Query (LINQ) continues to be one of my favorite features in all of .NET. As a combination of over 200 methods for performing various manipulations of arbitrary collections of data, it’s a succinct and flexible way to manipulate information. For example, during our work to optimize **Regex**, we’d frequently want to ask questions like “if I make this change to our loop auto-atomicity logic, how many loops could we now make atomic automatically,” and with a database of around 20,000 real-world regular expressions in hand, we could write LINQ queries against the internal object model used to represent a parsed regular expression “node tree” in order to quickly get the answer, like this:

int count =

 (from pattern in patterns

 let tree = Parse(pattern)

 from node in tree.EnumerateAllNodes()

 where node.Kind is

 RegexNodeKind.Oneloopatomic or

 RegexNodeKind.Notoneloopatomic or

 RegexNodeKind.Setloopatomic

 select node)

 .Count();

As LINQ is used in so many applications by so many developers, we strive to make it as efficient as possible (even though in our core libraries we still avoid using it on hot paths due to overheads involved in things like enumerator allocation and delegate invocation). Previous releases of .NET saw some significant improvements, for example due to reducing the algorithmic complexity of various operations by passing additional information from one operator to another (such as enabling an **OrderBy(…).ElementAt** to perform a “quick select” rather than “quick sort” operation). In .NET 7, one of the larger improvements in LINQ relates to a much larger set of optimizations throughout .NET 7, that of vectorization.

Vectorization is the process of changing an implementation to use vector instructions, which are SIMD (single instruction multiple data) instructions capable of processing multiple pieces of data at the same time. Imagine that you wanted to determine whether an array of 1,000,000 bytes contained any zeros. You could loop over every byte in the array looking for 0, in which case you’d be performing 1,000,000 reads and comparisons. But what if you instead treated the array of 1,000,000 bytes as a span of 250,000 **Int32** values? You’d then only need to perform 250,000 read and comparison operations, and since the cost of reading and comparing an **Int32** is generally no more expensive than the cost of reading and comparing a byte, you’d have just quadrupled the throughput of your loop. What if you instead handled it as a span of 125,000 **Int64** values? What if you could process even more of the data at a time? That’s vectorization.

Modern hardware provides the ability to process 128 bits, 256 bits, even 512 bits at a time (referred to as the width of the vector), with a plethora of instructions for performing various operations over a vector of data at a time. As you might guess, using these instructions can result in absolutely massive performance speedups. Many of these instructions were surfaced for direct consumption as “hardware intrinsics” in .NET Core 3.1 and .NET 5, but using those directly requires advanced know-how and is only recommended when absolutely necessary.

Higher level support has previously been exposed via the **Vector<T>** type, which enables you to write code in terms of **Vector<T>,** and the JIT then compiles that usage down to the best available instructions for the given system. **Vector<T>** is referred to as being “variable width,” because, depending on the system, the code actually ends up running on, it might map to 128-bit or 256-bit instructions, and because of that variable nature, the operations you can perform with it are somewhat limited. .NET 7 sees the introduction of the new fixed-width **Vector128<T>** and **Vector256<T>** types, which are much more flexible. Many of the public APIs in .NET itself are now vectorized using one or more of these approaches.

Some of LINQ was previously vectorized. In .NET 6, the **Enumerable.SequenceEqua**l method was augmented to special-case **T[]**, in which case, the implementation would use a vectorized implementation to compare the two arrays. In .NET 7, some of the overloads of Enumerable.Min, Enumerable.Max, Enumerable.Average, and Enumerable.Sum have all been improved.

First, these methods now all specialize for the very commonly used types **T[]** and **List<T>**, in order to optimize the processing of their contents. Both of these types make it easy to get a **ReadOnlySpan<T**> for their contents, which in turn means the contents of either can be processed with one shared routine that has fast access to each element rather than needing to go through an enumerator.

And then some of the implementations are able to take it further and vectorize that processing. Consider **public static double Enumerable.Average(this IEnumerable<int>)**, for example. Its behavior is to sum all of the **Int32** values in the source, accumulating the sum into an **Int64,** and then dividing that **Int64** by the number of summed elements (which needs to be at least 1). This can lead to huge speedups, for example:

using BenchmarkDotNet.Attributes;

using BenchmarkDotNet.Running;

using System.Buffers;

using System.Linq;

public partial class Program

{

 static void Main(string[] args) =>

 BenchmarkSwitcher.

 FromAssembly(typeof(Program).Assembly).

 Run(args);

 private int[] \_values =

 Enumerable.Range(0, 1\_000\_000).

 ToArray();

 [Benchmark]

 public double Average() =>

 \_values.Average();

}

On my computer, this shows the benchmark on .NET 6 taking an average of 3661.6us and on .NET 7 taking an average of 141.2us—almost 26 times faster. How would you vectorize an operation like this if you wanted to do it by hand? Let’s start with the sequential implementation, and for simplicity, let’s assume the input has already been validated to be non-null and non-empty.

static double Average(ReadOnlySpan<int> values)

{

 long sum = 0;

 for (int i = 0; i < values.Length; i++)

 {

 sum += values[i];

 }

 return (double)sum / values.Length;

}

The basic idea is that you want to process as much as possible vectorized, and then use the same one-at-a-time loop to process any remaining items. Start by validating that you’re running on hardware that can, in fact, accelerate the vectorized implementation and that you have enough data to vectorize at least something:

if (Vector.IsHardwareAccelerated &&

 values.Length >= Vector<int>.Count)

{

 ...

}

Once you know that, you can write your vectorized loop. The approach will be to maintain a **Vector<long>** of partial sums; for each **Vector<int>** you read from the input, you’ll “widen” it into two **Vector<long>**s, meaning every **int** will be cast to a **long**, and because that doubles the size, you’ll need two **Vector<long>**s to store the same data as the **Vector<int>.** Once you have those two **Vector<long>**s, you can just add them to your partial sums. And at the end, you can sum all of the partial sums together. That gives you this as your entire implementation:

static double Average(ReadOnlySpan<int> values)

{

 long sum = 0;

 int i = 0;

 if (Vector.IsHardwareAccelerated &&

 values.Length >= Vector<int>.Count)

 {

 Vector<long> sums = default;

 do

 {

 Vector.Widen(

 new Vector<int>(values.Slice(i)),

 out Vector<long> low,

 out Vector<long> high);

 sums += low;

 sums += high;

 i += Vector<int>.Count;

 }

 while (i <= values.Length –

 Vector<int>.Count);

 sum += Vector.Sum(sums);

 }

 for (; i < values.Length; i++)

 {

 sum += values[i];

 }

 return (double)sum / values.Length;

}

If you look at the implementation of this overload in .NET 7 (it’s open source, and you’re encouraged to read and contribute), you’ll see this is almost exactly what the official code currently does.

## Call to Action

.NET 7 is full of these kinds of performance improvements, across the entirety of the release. Please download .NET 7, upgrade your apps and services, and try it out for yourself. We’re excited to hear how these improvements contribute to the performance of your applications.

Stephen Toub

stoub@microsoft.com

Stephen Toub is a Partner Software Engineer at Microsoft. He spends most of his time focused on the libraries that comprise .NET and on performance across the .NET stack.