

A Deep Dive into Improving ASP.Net Core Performance

What are we going to cover?

We'll discuss the following points in the sections that follow:

- Use Caching to Improve Performance 4
- Use Structs in lieu of Classes 8
- Use Asynchronous Programming to Improve Performance 9
- Minimize Virtual Calls 10
- Minimize or Reduce Allocations 12
- Minimize Large Object Allocations 15
- Enable Response Compression 16
- Pool HTTP connections with HttpClientFactory 17
- Optimize CPU Cache Usage 18
- Optimize Data Access 20
- Optimize Disk Access 21
- Other performance issues in ASP.NET Core 22

Introduction

Performance is always important when building web applications, but it is especially important when using ASP.NET Core. The framework has been designed from the ground up with performance in mind, and as such, provides several features and capabilities that can help you improve the performance of your applications.

ASP.NET Core is a powerful web development framework, but like any tool, it has its own performance quirks that can trip up even the most experienced developers.

In this article, we'll take a deep dive into ASP.NET Core performance, examining some of the most significant issues and how we can solve them. We'll also look at some best practices that can help you avoid performance bottlenecks in the first place.



Cache Aggressively

Caching is one of the most important aspects of performance optimization. By caching data, we can avoid expensive roundtrips to the database or other data sources. In ASP.NET Core, you can cache your data in several ways.

The easiest way to cache data is to use the in-memory cache provided by the `Microsoft.Extensions.Caching.Memory` NuGet package that adds an in-memory cache service for use in ASP.NET Core applications.

To use the in-memory cache service, add the following line to your Program class:

```
services.AddMemoryCache();
```


Once you've added the memory cache service, you can inject an `IMemoryCache` instance into your controllers and services using constructor injection and use it to store and retrieve data from the cache:

```
public class HomeController : Controller
{
    private readonly IMemoryCache _cache;

    public HomeController(IMemoryCache cache)
    {
        _cache = cache;
    }
}
```

You can now use the `Set` method of the `IMemoryCache` instance to store data and retrieve data from the cache using the key used in the `Set` method:

```
_cache.Set("key", "value"); // Store data in the cache
var data = _cache["key"]; // Retrieve data from the cache
```



You can use the following piece of code to configure sliding and absolute configuration, set the cache priority and size of the cache and then add data to the cache.

```
var cacheEntryOptions = new MemoryCacheEntryOptions()
    .SetSlidingExpiration(TimeSpan.FromSeconds(90))
    .SetAbsoluteExpiration(TimeSpan.FromSeconds(1200))
    .SetPriority(CacheItemPriority.Normal)
    .SetSize(1024);
_cache.Set(employeeListCacheKey, data, cacheEntryOptions);
```

You can use the TryGetValue method to check if the specified key is available in the cache.

```
if (!cache.TryGetValue<string>
("myCacheKey", out string timestamp))
{
    cache.Set<string>("myCacheKey", DateTime.Now.ToString());
}
```

Besides storing and retrieving data from the in-memory cache, you can also use it to monitor when items are added or removed from the cache.

You can register a callback method that would be called automatically when an item is removed from the cache as shown in the code snippet given below:

```
MemoryCacheEntryOptions options =
new MemoryCacheEntryOptions();
options.AbsoluteExpiration = DateTime.Now.AddMinutes(1);
options.SlidingExpiration = TimeSpan.FromMinutes(1);
options.RegisterPostEvictionCallback(MyCacheCallback, this);
cache.Set<string>("myCacheKey", DateTime.Now.ToString(), options);
```

You can then specify the ResponseCache attribute on your action methods and set appropriate headers to the cached responses.

```
[ResponseCache(Duration = 60)]
public IActionResult Index()
{
    ViewData["myViewBagKey"] = "The current time is: " +
    DateTime.Now.ToString();
    return View();
}
```

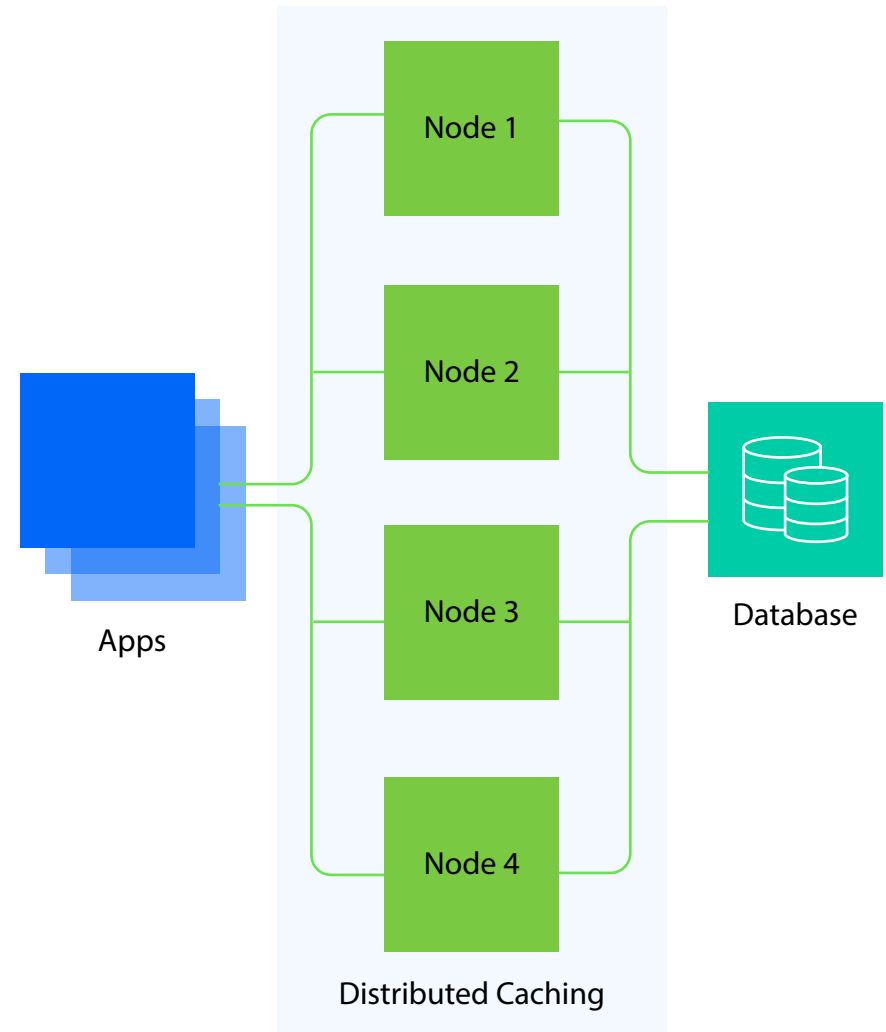
Distributed Caching

The in-memory cache is great for caching data that does not need to be persisted across application restarts. For data that does need to be persisted, you can use the distributed cache service provided by the `Microsoft.Extensions.Caching.Redis` NuGet package. The Redis distributed cache stores cached data in a Redis database, which can be run locally or hosted in the cloud. To use the Redis distributed cache, add the following line to your `ConfigureServices` method:

```
services.AddDistributedRedisCache(options => { options.Configuration =  
    "localhost"; options.InstanceName = "SampleInstance"; });
```

Once you've added the Redis distributed cache service, you can inject an `IDistributedCache` instance into your controllers and services using constructor injection:

```
public class HomeController : Controller { private readonly  
    IDistributedCache _cache; public HomeController  
    (IDistributedCache cache) { _cache = cache; } }
```



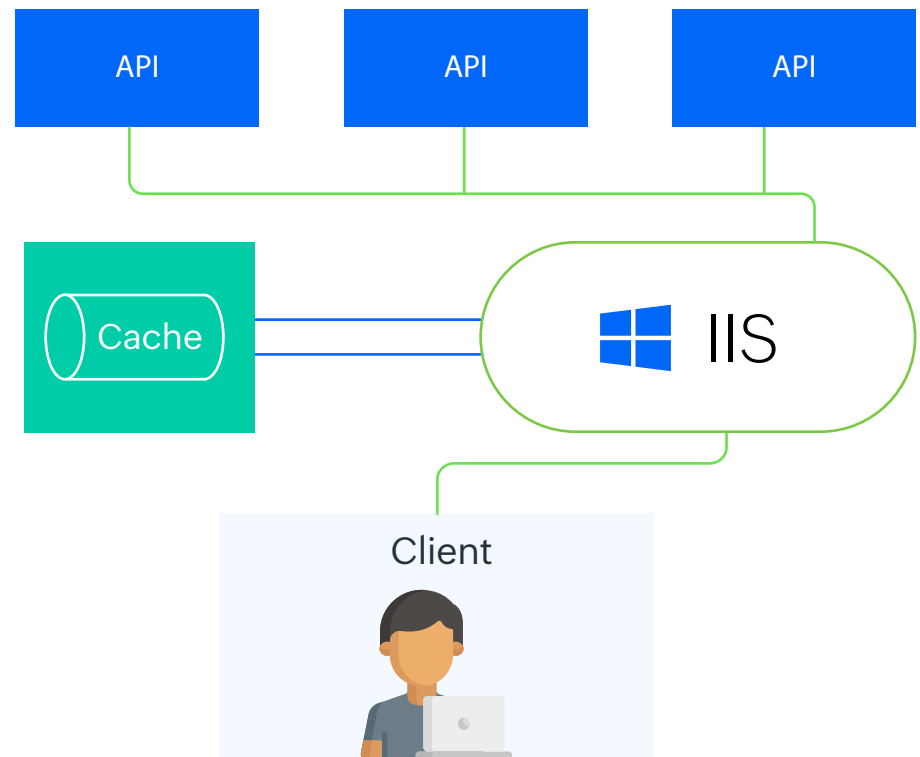
Output Caching

To improve ASP.NET Core performance, you should cache data aggressively. By caching data, you can reduce the number of round trips to the database and improve the response time of your web application. You can write the following piece of code in your Program class to enable response caching in your application:

```
var builder = WebApplication.CreateBuilder(args);  
builder.Services.AddControllers();  
builder.Services.AddResponseCaching();  
var app = builder.Build();  
app.UseHttpsRedirection();  
app.UseResponseCaching();  
app.UseAuthorization();  
app.MapControllers();  
app.Run();
```

You can also configure response caching to specify the size limit, maximum body size, and if the HTTP responses should be cached on sensitive paths:

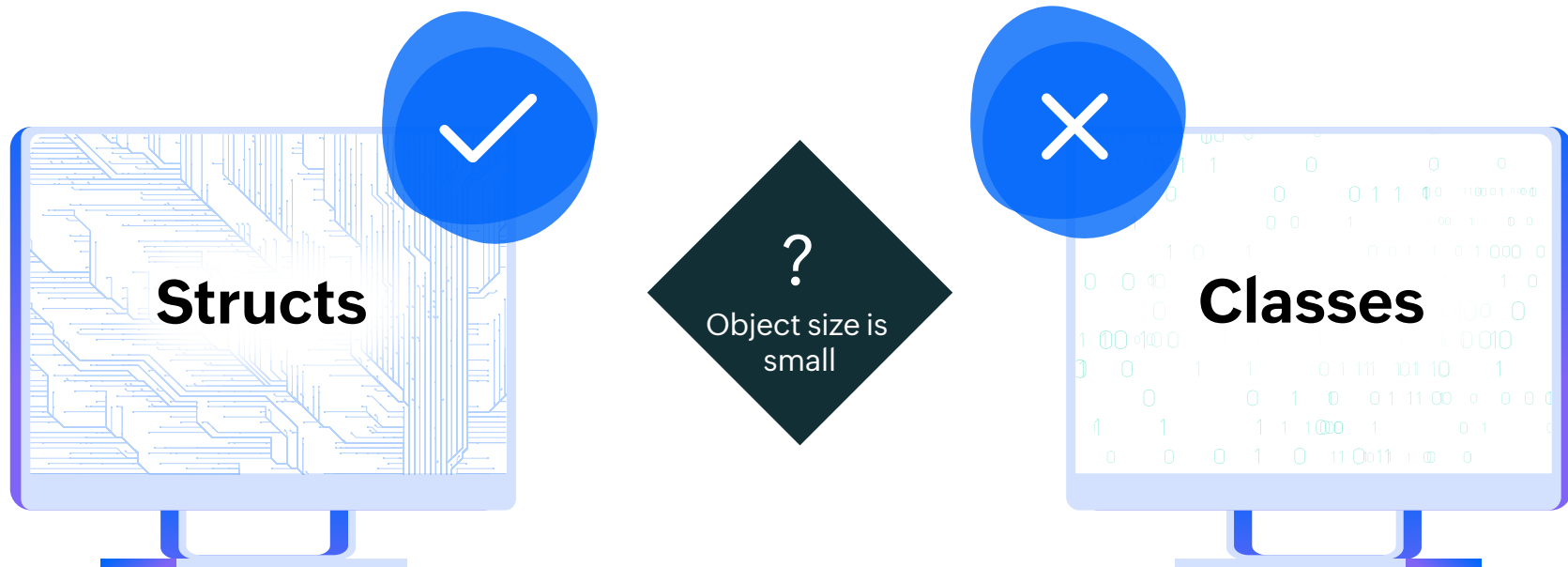
```
services.AddResponseCaching(options =>  
{  
    options.UseCaseSensitivePaths = true;  
    options.MaximumBodySize = 1024;  
});
```



Use Structs in lieu of Classes

In the C# programming language, value and reference types' semantics are very different. A reference type resides on the heap and is passed by reference, while a value type resides on the stack and is passed by value (e.g., the receiving method receives a copy of the whole object).

The cost of allocating and deallocating value types is lower than reference types. You can avoid garbage collection overhead by using a struct when building a composite data structure with a few data members and value semantics. Ideally, the instance size of a struct should be less than 16 bytes.



Use Asynchronous Programming

You can leverage asynchronous programming to execute multiple tasks concurrently without blocking the processing of other tasks. On the contrary, synchronous programming, i.e., code written using the synchronous programming model, blocks until a specific task is completed.

The following code snippet shows how an async method can be defined in C#:

```
public async Task<List<Employee>> GetEmployeeAsync()
{
    List<Employee> employees;

    using (var connection = new SqlConnection(connectionString))
    {
        await connection.OpenAsync();

        //Write your code here to read data from the database
    }

    return employees;
}
```

Asynchronous programming provides more flexibility than synchronous programming and allows multiple operations to occur simultaneously without waiting for each other—making it ideal for applications that do not require immediate responses from users but still need to perform many operations at once (such as image processing).

Asynchronous programming is a great way to keep your application responsive by reducing the time required for the server to generate a response, and thus keeping your application responsive. You can use `async/await` methods to write asynchronous code that's easier to read than traditional async programming. Async programming is used to overcome the limitations of the synchronous programming model and make your code more efficient and responsive.

Asynchronous programming is a more efficient way to handle operations that take a long time. This is because the process doesn't have to wait for the operation to complete before moving on. Instead, it can work on other things while the operation runs in the background. This is especially important for web applications because they are often processing many requests at once and must handle each one as quickly as possible.

Minimize Virtual Calls

Virtual calls are expensive because they require a bit of extra processing by the computer in order to work. When you make a virtual call, the runtime has to look up the method in the virtual table, and then jump to that code. This takes time, and it can accumulate if you're making a lot of calls. You can eliminate such virtual calls using interfaces.

Virtual approaches need additional resources. A virtual table known as vtable maps all the virtual methods of a class. This table lists all the virtual methods defined in a class. Virtual methods may be slow since the runtime must verify the object's type every time they're called. When you call a virtual method on an object, the runtime searches the vtable for metadata related to the virtual methods of the class to which the object belongs.

Interfaces don't have virtual tables, so the runtime doesn't have to do any extra work when you make a call through an interface. Additionally, you can make your methods static to avoid using virtual methods. Now, consider the following piece of code that takes advantage of virtual methods to implement runtime polymorphism. Runtime polymorphism (also known as Dynamic Method Dispatch or dynamic binding) resolves the call to an overridden method only at runtime.

```
public abstract class Shape
{
    protected virtual void Draw()
    {
        Console.WriteLine("Draw method of the Shape class called.");
    }
}
class Circle : Shape
{
    protected override void Draw()
    {
        Console.WriteLine("Draw method of the Circle class called.");
    }
}
class Rectangle : Shape
{
    protected override void Draw()
    {
        Console.WriteLine("Draw method of the Rectangle class called.");
    }
}
```

You can replace the preceding code listing with the one that follows:

```
public interface IShape
{
    protected void Draw()
    {
        Console.WriteLine("Draw method of the Shape class called.");
    }
}
class Circle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Draw method of the Circle class called.");
    }
}

class Rectangle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Draw method of the Rectangle class called.");
    }
}
```

Using interfaces, you can simulate the same behaviour but sans the overhead cost of tables.

Minimize or Reduce Allocations

When it comes to improving performance, one area that is often overlooked is reducing allocations. Allocations are a necessary part of any application, but they can have a significant impact on performance if not managed properly.

There are a few ways to reduce allocations in ASP.NET Core applications:

- Use the `Span<T>` type for operations that don't require copying data.
- Avoid using async methods unnecessarily.
- Use `ArrayPool<T>` for pooled arrays.
- Use `String.Create` to create strings with zero allocation overhead.
- Use `StringBuilder` for string concatenation.
- Avoid LINQ methods that cause unnecessary allocations.

It is worth taking a closer look at each of them one at a time.

Use the Span type for operations that don't require copying data

The Span type was introduced in .NET Standard 2.0 and is available in ASP.NET Core 2.0 and above. It embodies a contiguous region of memory, and can be used as an alternative to arrays or `List<T>`.

One advantage of `Span<T>` over arrays is that it doesn't require copying data when performing operations such as filtering or mapping.

`Span<T>` is defined as a struct as shown in the code snippet given below:

```
public readonly ref struct Span<T>
{
    internal readonly
    ByReference<T> _pointer;
    private readonly int _length;
    //Other members
}
```

The following code snippet shows how you can leverage `Span<T>` to extract a slice of two elements from an array that comprises prime numbers:

```
int[] primes = new int[] { 1, 2, 3, 5, 7, 11 };
Span<int> slicedArray = new Span<int>(primes, 1, 2);
foreach(int i in slicedArray)
{
    Console.WriteLine(i);
}
```

Avoid using async methods unnecessarily

If the `async` keyword is used in the code, the compiler generates a state machine that allows the method to be executed asynchronously instead of synchronously. Since using this state machine incurs a significant amount of performance overhead, it is advised to use it only when necessary.

In general, you should avoid using `async` methods unless they're actually going to perform an asynchronous operation. For example, the following code uses an `async` method to perform a synchronous operation:

```
public async Task<int> Add(int x, int y)
{
    return x + y; //this operation is synchronous
}
```

This code will incur the overhead of the state machine without any benefits. It would be more efficient to just remove the `async` keyword as shown in the code snippet given below:

```
public Task Add(int x, int y)
{
    return Task.FromResult(x + y);
}
```

Use ArrayPool for pooled arrays

The `ArrayPool` class was introduced in .NET Standard 2.0 and is available in ASP.NET Core 2.0 and above. It allows arrays to be pooled and reused, which can reduce allocations significantly.

For example, consider the following code that allocates an array to store a list of numbers:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int[] numbersArray = numbers.ToArray(); //allocates a new array
```

This code will allocate a new array every time it's called. We can avoid this by using `ArrayPool`:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int[] numbersArray = ArrayPool.Shared.Rent(numbers.Count); //rents
an array from the pool
numbers.CopyTo(numbersArray); //copies the data into the array
ArrayPool.Shared.Return(numbersArray); //returns the array to
the pool
```

Use `String.Create` to create strings without any allocation overhead

`String.Create` is a method that allocates memory on the managed heap to store a sequence of characters. The following code snippet shows how you can use `String.Create` in C# to create string objects:

```
char[] buffer = { 'a', 'b', 'c', 'd', 'e' };
string data = string.Create(buffer.Length, buffer, (x, y) => {
    for (int i = 0; i < x.Length; i++)
        x[i] = y[i];
});
```

Use `StringBuilder` for string concatenation

The `StringBuilder` class is used to efficiently concatenate strings. It avoids allocations by using a mutable buffer that can be reused.

For example, consider the following code that concatenates a list of strings:

```
List strings = new List { "a", "b", "c" };
string result = string.Concat(strings); //allocates memory for a new
string instance
```

This code will allocate a new string every time it's called. We can avoid this by using `StringBuilder`:

```
List strings = new List { "a", "b", "c" };
StringBuilder builder = new StringBuilder();
foreach (string s in strings)
{
    builder.Append(s);
}
string result = builder.ToString(); //returns the concatenated string
```

Avoid LINQ methods that cause unnecessary allocations

Some LINQ methods, such as `Select` and `Where`, will cause unnecessary allocations if the data type of the source sequence is not `IEnumerable`.

```
List<int> numbers = new List { 1, 2, 3, 4, 5 };
//The following piece of code will allocate a new list
IEnumerable evenNumbers = numbers.Where(n => n % 2 == 0);
```

This code will allocate a new `List`, even though the original `List` contains all the data we need. We can avoid this allocation by using the `AsEnumerable` extension method:

```
List numbers = new List { 1, 2, 3, 4, 5 };
```

```
//The following piece of code will not allocate a new list
IEnumerable evenNumbers = numbers.AsEnumerable().Where
(n => n % 2 == 0);
```

Minimize Large Object Allocations

The Garbage Collector (GC) divides objects into two categories – small objects and large objects. While the former is stored on the Small Object Heap (SOH), the latter is stored in the Large Object Heap (LOH). Large objects are those that have a size greater than or equal to 85,000 bytes. An object allocation request of 85,000 bytes or more is allocated to the large object heap by the runtime.

Note that garbage collection for large objects is a costly operation. If you are interested in reducing large object allocations, you can leverage pool buffers using an `ArrayPool<T>` and cache large objects that are often used. On frequent code paths, you should avoid getting locks on short-lived objects, and you should try to avoid creating too many short-lived objects along these paths.

Here are a few best practices in this regard:

- Consider caching large objects to save costly allocations.
- It would help if you took advantage of `ArrayPool<T>` to store large arrays.
- Avoid allocating large objects on hot code paths.
- Optimize code to split large collections into multiple smaller collections to avoid objects being created in the large object heap.
- Use `MemoryPool<T>` to reduce allocations.

The following code snippet shows how you can use `ArrayPool<T>` in C#:

```
var myArrayPool = ArrayPool<int>.Create(5, 10);  
var myRentedArray = myArrayPool.Rent(5);
```

Enable Response Compression

ASP.NET Core uses a middleware component to automatically compress server responses. This can greatly improve performance, especially over slow network connections. Enabling compression is easy and only requires a few lines of code:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression();

var app = builder.Build();

//Usual code to register services, dependencies, etc

app.UseResponseCompression();

app.MapGet("/", () => "Demonstrating Response Compression.");

app.Run();
```

Compression will now be enabled for all responses from your ASP.NET Core application.

Pool HTTP connections with HttpClientFactory

HttpClientFactory is a new feature in ASP.NET Core that makes it easy to inject a properly configured HttpClient instance into your controllers and services. By default, HttpClient instances are not pooled, which means that each time you create a new one, it will open a new connection to the server. If you make multiple requests to the same server, this can be costly as far as application performance is concerned. Fortunately, HttpClientFactory supports connection pooling out of the box. When you configure HttpClientFactory to use a pooling policy, it will reuse existing connections instead of creating new ones. This reduces the number of connections you need to be opened and closed in your application, thus improving performance. To configure HttpClientFactory to use a pooling policy, you first need to add the Microsoft.Extensions.Http NuGet package to your project. Then, you can use the AddHttpClient method on the IServiceCollection object as shown in the code snippet given below:

```
services.AddHttpClient();
```

This will configure HttpClientFactory to use the default pooling policy, which is to keep a maximum of 10 connections open per endpoint. You can customize this by passing in an Action to the AddHttpClient method:

```
builder.Services.AddHttpClient((httpClient) => {  
    httpClient.MaxConnectionsPerEndpoint = 20; }); }
```

Optimize CPU Cache Usage

You can leverage CPU cache to improve the performance of your application. When you're accessing data from main memory, you can take advantage of the CPU cache to minimize memory latency.

Here are a few points that should help you to make the most of the CPU cache:

- Utilize linear data structures based on access patterns instead of algorithms and data structures with irregular memory access patterns.
- Ensure that the data types are small and that it is structured in a way to prevent gaps in the alignment.
- Improve spatial locality by making the most of each cache line after mapping it to a cache.

We can increase data locality and reduce waste of CPU cycles by optimizing our code. Let us understand this with an example. Assume that there are two integer arrays and the elements of both these arrays have been initialized to 1. Now, suppose there are two methods named ScenerioA and ScenerioB that multiplies the value of each element of these two arrays by 10 and stores the result in those elements.

While the method ScenerioA uses one for loop to iterate and reinitialize the elements of these two arrays, the method ScenerioB uses separate loops to do the same. Consider the following piece of code that

illustrates what we just discussed:

```
internal class Program
{
    static Stopwatch stopWatch = new Stopwatch();
    const int LENGTH = 1000;
    static int[] integerArrayA = Enumerable.Repeat(1,
        LENGTH).ToArray();
    static int[] integerArrayB = Enumerable.Repeat(1,
        LENGTH).ToArray();
    private static void ScenerioA()
    {
        stopWatch.Restart();
        for (int i = 0; i < LENGTH; i++)
        {
            integerArrayA[i] *= 10;
            integerArrayB[i] *= 10;
        }
        stopWatch.Stop();
        TimeSpan ts = stopWatch.Elapsed;
        Console.WriteLine($"Scenerio A:
        {ts.TotalMilliseconds.ToString()} ms");
    }
}
```

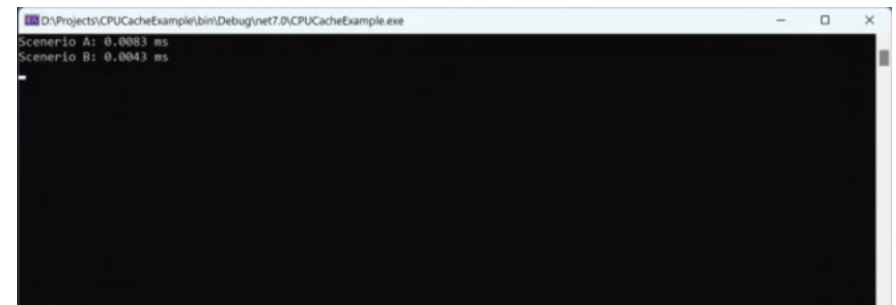
```

private static void ScenerioB()
{
    stopWatch.Restart();
    for (int i = 0; i < LENGTH; i++)
    {
        integerArrayA[i] *= 10;
    }
    for (int i = 0; i < LENGTH; i++)
    {
        integerArrayB[i] *= 10;
    }
    stopWatch.Stop();
    TimeSpan ts = stopWatch.Elapsed;
    Console.WriteLine($"Scenerio B:
    {ts.TotalMilliseconds.ToString()} ms");
}

static void Main(string[] args)
{
    ScenerioA();
    ScenerioB();
    Console.Read();
}
}

```

When you execute the preceding code listing, the time taken to run both scenarios will be displayed:



The screenshot shows a console window titled "D:\Projects\CPUCacheExample\bin\Debug\net7.0\CPUCacheExample.exe". The output text is as follows:

```

Scenerio A: 0.0083 ms
Scenerio B: 0.0043 ms

```

Figure 1

Note that the time taken by ScenerioA is almost double the time taken by the method ScenerioB to execute. The reason is that since the collections we've used is an array, the elements of this collection will be stored adjacent to one another. In case of ScenerioB, because most of the data is in the cache lines, it would consume less time to read the data from the element, perform a simple computation and then store the result back.

In case of ScenerioA, the elements of the first array will be stored in the cache. However, the data will not be available in the cache when your program attempts to access the second array. This would result in several cache misses and hence the result.

Optimizing Data Access

Here are a few best practices in this regard:

- Reduce the number of HTTP calls, i.e., network round trips.
- Instead of sending multiple calls to the server, consider retrieving the data in one or two calls and aggregating the data if required.
- Use caching for data that is stale.
- Avoid obtaining the data in advance.

Best Practices in Entity Framework Core for Improving Performance

- Ensure that your DataContext Represents a Single Unit of Work
- Disable Change Tracking if you're only reading data
- Use Projections to Read Only the Data You Need
- Use DbContextPool to improve EF Core performance and scalability

To turn on DbContextPool, you can write the following piece of code in your Program class:

```
services.AddDbContextPool<EmployeeContext>(options =>  
options.UseSqlServer(connection));
```

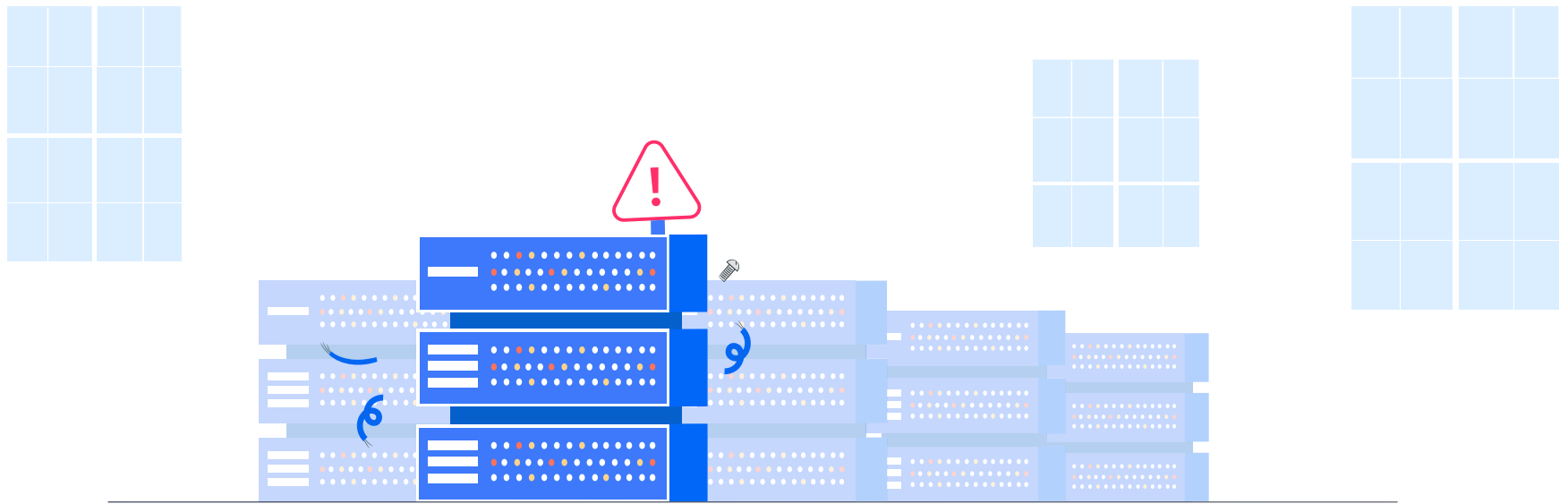


Optimizing Disk Access

In addition to optimizing data access, it is also important to optimize I/O operations. One way to do this is to use asynchronous programming techniques when possible. By using asynchronous programming, you can avoid blocking threads and improve overall performance. You can also improve I/O performance by leveraging buffering when reading from or writing to files in the file system. Buffering allows you to read or write larger chunks of data at a time, which can significantly improve performance.

The following code snippet illustrates how you can use buffered stream in C#:

```
services.AddDbContextPool<EmployeeContext>(options =>  
options.UseSqlServer(connection));
```



Other performance issues in ASP.NET Core

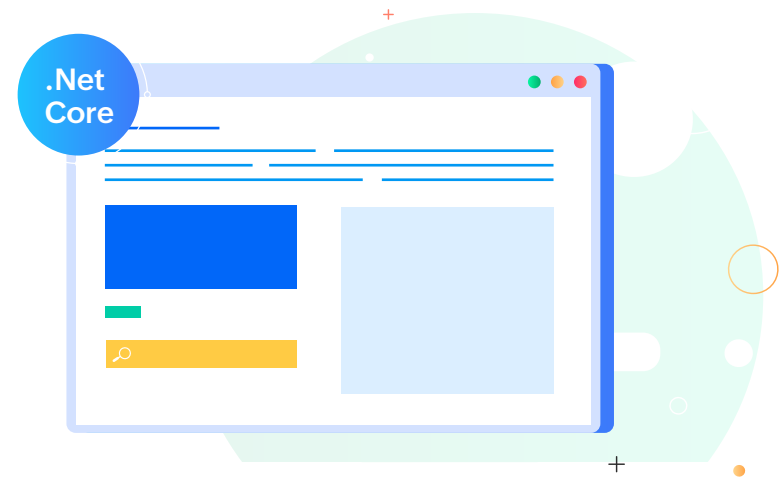
There are a few common performance issues that can occur when using ASP.NET Core. One issue is when the web server becomes overloaded and cannot process requests quickly enough. This can often happen during high traffic times or if there is a lot of data being processed. Another common issue is when the database server is not configured correctly and is not able to handle the load of requests. This can often lead to slow response times or even timeouts.

Hosting Model

One common issue that can impact ASP.NET Core performance is using the wrong hosting model. For example, if you're using IIS as your web server, you should make sure that you're using the IIS integrated pipeline mode rather than the legacy mode. This will ensure that your ASP.NET Core application benefits from all of the performance improvements in IIS such as kernel-mode caching and CPU limits.

Third-party dependencies

If your application is making too many calls to external services or libraries, it can negatively impact performance. To mitigate this, you should try to use async patterns wherever possible so that your application can continue to process requests even while waiting on a response from an external dependency.



Static files

ASP.NET Core performance can be affected by how your web server serves static files. By default, IIS will serve static files from your ASP.NET Core application using its own static file handler rather than letting ASP.NET Core handle them directly. This can impact performance because IIS's static file handler is not as efficient as ASP.NET Core's static file middleware.



Conclusion

ASP.NET Core is a high-performance framework and can give you the best application performance. However, there are some common performance issues that you need to be aware of when developing a web application that leverages ASP.NET Core.

There are a number of ways to improve the performance of an ASP.NET Core application. By implementing the techniques discussed here, you can ensure that your ASP.NET Core application runs faster and more efficiently.

About Site24x7

Site24x7 offers AI-powered full stack monitoring for DevOps and IT operations with telemetry data collected from servers, containers, networks, cloud, database, applications and provide AI-powered full stack observability. Additionally, Site24x7 can track end user experience via synthetic and real user monitoring capabilities. DevOps & IT teams can use these capabilities to troubleshoot and resolve application downtime and performance issues, infrastructure issues and better manage the digital user experience. For more information on Site24x7, please visit www.Site24x7.com |

Email: eval@site24x7.com

[Get Quote](#)

[Request demo](#)