Performance Optimization with Span<T> and Memory<T> in C# When Handling HTTP Requests: Real-World Examples and Approaches

Daniel Damyanov, Ivaylo Donchev
Department of Information Technologies, Veliko Tarnovo University, Bulgaria

Abstract—**Optimization of application performance is a critical aspect of software development, especially when dealing with high-throughput operations such as handling HTTP requests. In modern C#, structures Span<T> and Memory<T> provide powerful tools for working with memory more efficiently, reducing heap allocations, and improving overall performance. This paper explores the practical applications of Span<T> and Memory<T> in the context of optimizing HTTP request processing. Real-world examples and approaches that demonstrate how these types can minimize memory fragmentation are presented, avoid unnecessary data copying, and enable high-performance parsing and transformation of HTTP request data. By leveraging these advanced memory structures, developers can significantly enhance the throughput and responsiveness of their applications, particularly in resource-constrained environments or systems handling many concurrent requests. This paper aims to provide developers with actionable insights and strategies for integrating these techniques into their .NET applications for improved performance.**

Keywords—NET; C#; optimization; memory optimization; span; memory; development; HTTP requests; data structures

I. Introduction

As software applications evolve and network communications become increasingly complex, code efficiency and performance are becoming key. In .NET, types such as Span<T> [1] and Memory<T> [2] provide powerful memory management and data processing tools that can play an essential role in optimizing performance when handling HTTP requests [3]. Efficient memory management is a critical concern in modern high-performance applications, especially when dealing with high volumes of HTTP requests and responses. In scenarios such as web servers, API gateways, and microservices, where throughput and responsiveness are paramount, excessive memory allocations and garbage collection (GC) [4] overhead can severely impact performance and scalability. Traditional approaches to handling HTTP data, like using StreamReader or working directly with strings and arrays, often lead to unnecessary memory copying and fragmentation, which increases the load on the garbage collector (GC) and reduces the overall efficiency of the system. To address these issues, modern versions of C# introduced Span<T> and ReadOnlySpan<T> – lightweight, stack-allocated types designed to operate directly on memory without allocations. These types allow developers to access and manipulate slices of memory, buffers or arrays without creating new objects, thereby minimizing memory allocations and avoiding costly GC operations. Span<T> enables mutable operations on memory, while ReadOnlySpan<T> provides safe, immutable access, making them ideal for handling both mutable and immutable HTTP request and response data. This paper explores how these advanced memory constructs can be applied in real-world HTTP request handling scenarios to optimize performance. We demonstrate practical examples of using Span<T> and ReadOnlySpan<T> to efficiently parse, modify, and process HTTP request and response data, highlighting the performance improvements gained from reduced memory usage and minimized garbage collection. By leveraging these tools, developers can significantly improve the throughput and scalability of their .NET applications, particularly in resource-constrained or high-load environments.

The following sections are organized as follows:

Section II comments on typical use cases of the Span and Memory structures. It points out their advantages, proven by test results. Emphasis is placed on the optimization of HTTP requests, sorting algorithms and parallel data processing.

Section III summarizes the comparison of the different memory management techniques in C#.

Section IV discusses the results of similar studies; focuses on the advantages of new memory management structures, including reducing memory allocation and optimizing GC operations. Attention is paid to trade-offs and limitations related to the use of Span and Memory, practical implications for HTTP request handling and similar features in other programming languages.

The conclusion motivates once again developers to use the new memory management structures to achieve higher performance of their applications.

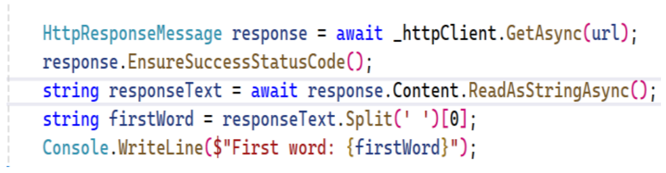
II. Common Use Cases of Span and Memory in Applications and Real Tests

When working with large text data, like reading content from files or HTTP responses, the traditional approach involves loading all the content into memory as a string. This can lead to a significant memory load, especially when the data is large. One of the common problems when dealing with HTTP requests is the incorrect handling of large JSON responses [5]. When receiving a large JSON response from a server, trying to desearilize the entire response at once can lead to memory and performance problems.

The following example shows one such problematic JSON response that is handled incorrectly, and this leads to a large memory load.

One common problem when dealing with data from HTTP requests is the inefficient handling of large text data, which can lead to unnecessary data copying and increased memory usage.

Span<T> can help optimize data processing by allowing us to work directly on arrays of bytes without copying. Span<T> comprises just two fields, a pointer and a length. For this reason, it can represent only contiguous blocks of memory [6]. Span by nature is mutable and allows for modifications to the underlying data. This example shows how a large text response is handled inefficiently, resulting in high memory overhead and latency.



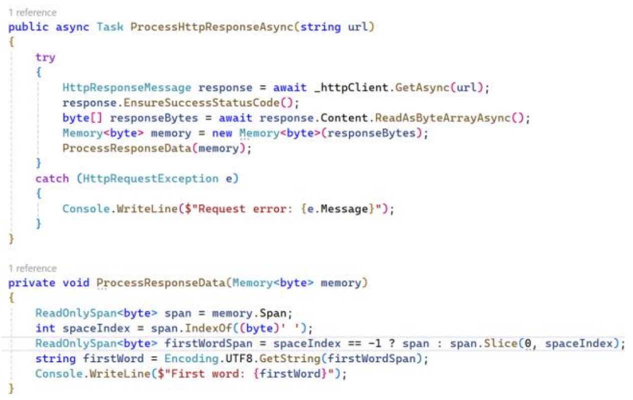
```
HttpResponseMessage response = await _httpClient.GetAsync(url);
response.EnsureSuccessStatusCode();
string responseText = await response.Content.ReadAsStringAsync();
string firstWord = responseText.Split(' ')[0];
Console.WriteLine($"First word: {firstWord}");
```

Fig. 1. Getting a response and taking the result from the null index.

The problem with the code above (Fig. 1) is the large and inefficient use of memory: the entire textual response is held in memory as a string. Therefore, unnecessary data copying is also performed. The Split() method creates new arrays of strings. To avoid these problems, Span<T> can be used to work directly on the byte arrays and to extract the necessary information efficiently. Using Span is only synchronous, and it should be noted that implementation in asynchronous methods requires additional code writing, since using synchronous methods in asynchronous ones would lead to unpredictable results or thread blocking.

Avoiding asynchronous disadvantages, another fast data processing can be used and the use of asynchronous operations without accompanying difficulties that can be achieved using Memory<T>, which allows working with subsets of data without additional copying.

Processing large text data from HTTP responses can be inefficient if the data is behaved like strings. Memory<T> can help optimize this process by working directly with the byte arrays. In .NET, Memory<T> and ReadOnlyMemory<T> provide powerful memory tools that enable efficient data processing without unnecessary copying and reduce memory load. These structures can be used in both synchronous and asynchronous methods (Fig. 2), making them suitable for a wide range of applications.



```
1 reference
public async Task ProcessHttpResponseAsync(string url)
{
    try
    {
        HttpResponseMessage response = await _httpClient.GetAsync(url);
        response.EnsureSuccessStatusCode();
        byte[] responseBytes = await response.Content.ReadAsByteArrayAsync();
        Memory<byte> memory = new Memory<byte>(responseBytes);
        ProcessResponseData(memory);
    }
    catch (HttpRequestException e)
    {
        Console.WriteLine($"Request error: {e.Message}");
    }
}

1 reference
private void ProcessResponseData(Memory<byte> memory)
{
    ReadOnlySpan<byte> span = memory.Span;
    int spaceIndex = span.IndexOf((byte)' ');
    ReadOnlySpan<byte> firstWordSpan = spaceIndex == -1 ? span : span.Slice(0, spaceIndex);
    string firstWord = Encoding.UTF8.GetString(firstWordSpan);
    Console.WriteLine($"First word: {firstWord}");
}
```

Fig. 2. Use of memory in http requests.

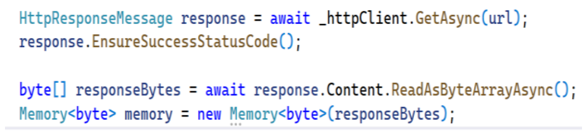
A. *Benefits of Memory<T> and ReadOnlyMemory<T>*

- Efficient memory usage: Memory<T> and ReadOnlyMemory<T> allows processing of parts of arrays without creating new objects, which reduces memory load.
- Flexibility: They can be used in both synchronous and asynchronous methods, providing a safe way to work with data between await points.
- Improved data processing code: By using Memory<T>, efficiency of applications can be improved when dealing with large amounts of data, such as text responses from HTTP requests or binaries.

B. *Test Performance When Reading Query Data*

When working with large files, it is often necessary to read only part of the content to avoid unnecessary memory load. Big data can fill up a system's RAM, leading to delays or crashes (OutOfMemoryException) [7]. Also, big data leads to slow processing speed [8] – a significant amount of time to reduce the efficiency of applications. Big data can contain unstructured or poorly formatted parts, which can complicate its processing.

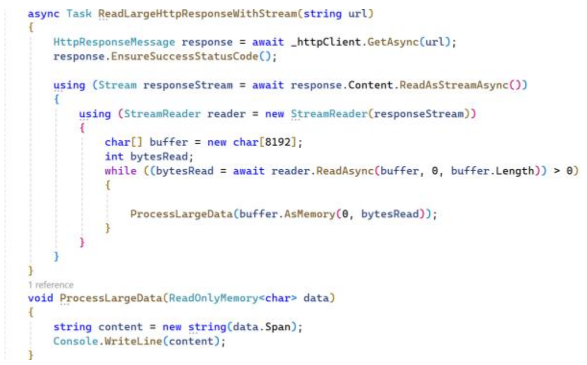
Using Memory<T> allows working with parts of the data without loading the entire file into memory. Reading a specific block of the file and using Memory<T> to work with that block provides efficiency and flexibility (Fig. 3).



```
HttpResponseMessage response = await _httpClient.GetAsync(url);
response.EnsureSuccessStatusCode();

byte[] responseBytes = await response.Content.ReadAsByteArrayAsync();
Memory<byte> memory = new Memory<byte>(responseBytes);
```

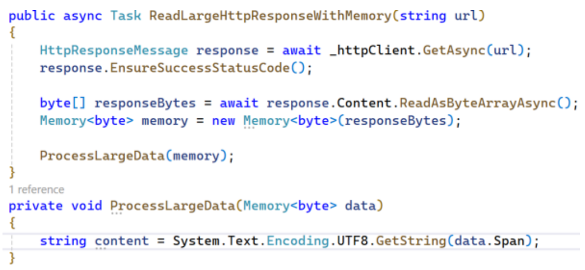
Fig. 3. Reading data with Memory<T>

The following is a comparative analysis between the use of Memory<T> and the standard implementation with StreamReader. Two methods will be implemented to read the response from the query (Fig. 4, Fig. 5).



```
async Task ReadLargeHttpResponseWithStream(string url)
{
    HttpResponseMessage response = await _httpClient.GetAsync(url);
    response.EnsureSuccessStatusCode();

    using (Stream responseStream = await response.Content.ReadAsStreamAsync())
    {
        using (StreamReader reader = new StreamReader(responseStream))
        {
            char[] buffer = new char[8192];
            int bytesRead;
            while ((bytesRead = await reader.ReadAsync(buffer, 0, buffer.Length)) > 0)
            {

                ProcessLargeData(buffer.AsMemory(0, bytesRead));
            }
        }
    }
}
1 reference
void ProcessLargeData(ReadOnlyMemory<char> data)
{
    string content = new string(data.Span);
    Console.WriteLine(content);
}
```

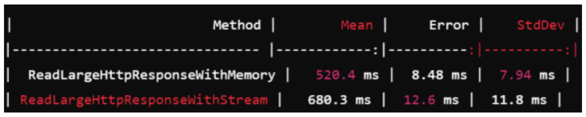
Fig. 4. Reading data with StreamReader.



```
public async Task ReadLargeHttpResponseWithMemory(string url)
{
    HttpResponseMessage response = await _httpClient.GetAsync(url);
    response.EnsureSuccessStatusCode();

    byte[] responseBytes = await response.Content.ReadAsByteArrayAsync();
    Memory<byte> memory = new Memory<byte>(responseBytes);

    ProcessLargeData(memory);
}
1 reference
private void ProcessLargeData(Memory<byte> data)
{
    string content = System.Text.Encoding.UTF8.GetString(data.Span);
}
```

Fig. 5. Reading data with Memory<T>.

HttpClient and URL were used for the tests. The URL points to a large file (100 MB) which can be downloaded for the tests. Running the benchmark leads to the following results (Fig. 6):



```
|                        Method |       Mean |     Error |    StdDev |
|------------------------------ |-----------:|----------:|----------:|
|  ReadLargeHttpResponseWithMemory |   520.4 ms |  8.48 ms |  7.94 ms |
| ReadLargeHttpResponseWithStream |   680.3 ms |  12.6 ms |  11.8 ms |
```

Fig. 6. Test results.

- ReadLargeHttpResponseWithMemory(): This method uses Memory<byte> and shows an average execution time of 520.4 ms. Using Memory<T> provides advantages in terms of efficient memory management and performance.

- ReadLargeHttpResponseWithStream(): This method uses a traditional approach with Stream and shows an average execution time of 680.3 ms. The traditional method is slower due to the additional time it takes to work with StreamReader and convert data.

Several of the most important advantages of using relevant optimizations can be noted:

- Efficient Memory Management: Working with Memory<byte> reduces the need to create redundant copies of data, which saves memory and resources.

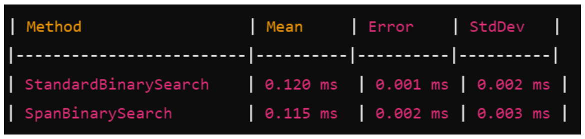
- Flexibility: Memory<T> allows for easy retrieval and handling of partial blocks of data, which is useful when working with large files or data streams.

- Better performance: Reading only the required portion of the file and working with Memory<T> can improve application performance by reducing processing time and memory usage.

C. *Other applications*

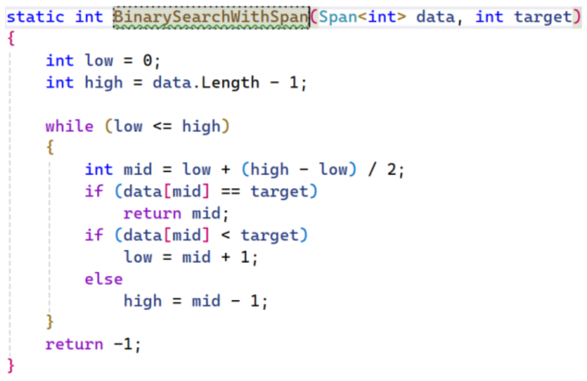
1) Search algorithm optimization: Algorithms for searching large arrays or text data often require high performance and low memory usage. Traditional approaches may involve creating new copies of the data, which increases memory costs and slows down processing. Span<T> allows working with pieces of data directly, without creating new copies, which is useful in search algorithm optimization. The following test will be conducted. A version of the familiar binary search algorithm [9] is implemented but using the Span structure. It must be compared to a method representing the basic implementation. Finally, the test is run again.

Fig. 8 shows the only difference is that the lookup data is sent with a Span. In the benchmark made, in Fig. 7, 1 million elements were feeded.



```
| Method                 | Mean     | Error    | StdDev   |
|------------------------|----------|----------|----------|
| StandardBinarySearch   | 0.120 ms  | 0.001 ms | 0.002 ms |
| SpanBinarySearch       | 0.115 ms  | 0.002 ms | 0.003 ms |
```

Fig. 7. Binary search and span.



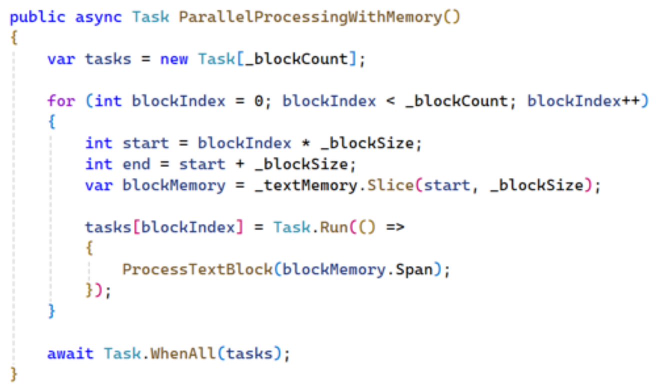
```
static int BinarySearchWithSpan(Span<int> data, int target)
{
    int low = 0;
    int high = data.Length - 1;

    while (low <= high)
    {
        int mid = low + (high - low) / 2;
        if (data[mid] == target)
            return mid;
        if (data[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

Fig. 8. Binary search with a span.

It is noticeable that the difference is small, but it will be tested whether Span<T> offers any advantages in the context of specific use cases. One of the advantages is that ready-made methods of the structure itself can be used, and additional functionality can be added without wasting time on it.

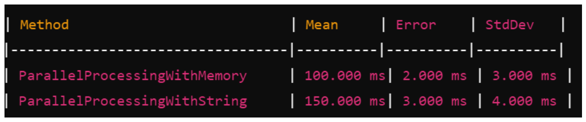
2) Parallel data processing: Parallel data processing can lead to high memory costs if each thread creates its own copies of the data. This can reduce the efficiency of the application and increase processing time. Parallel processing involves dividing a task into smaller subtasks that can be run simultaneously [10]. C# has various tools for parallelization., such as Task, Parallel.For, and async/await. The following test will be performed. 10 million characters simulating text will be sent to be processed by the program. Two variants have been developed: with string and with Memory (Fig. 9 and Fig. 11).

In the test done, better performance is again visible, albeit with a small lead in time (Fig. 10).



```
public async Task ParallelProcessingWithMemory()
{
    var tasks = new Task[_blockCount];

    for (int blockIndex = 0; blockIndex < _blockCount; blockIndex++)
    {
        int start = blockIndex * _blockSize;
        int end = start + _blockSize;
        var blockMemory = _textMemory.Slice(start, _blockSize);

        tasks[blockIndex] = Task.Run(() =>
        {
            ProcessTextBlock(blockMemory.Span);
        });
    }

    await Task.WhenAll(tasks);
}
```

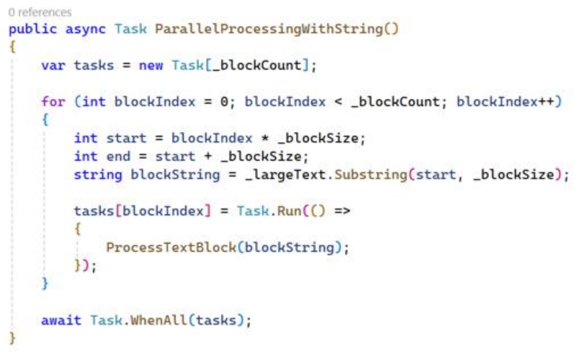
Fig. 9. Parallel processing with memory



```
| Method                         | Mean      | Error     | StdDev    |
|--------------------------------|-----------|-----------|-----------|
| ParallelProcessingWithMemory   | 100.000 ms| 2.000 ms  | 3.000 ms  |
| ParallelProcessingWithString   | 150.000 ms| 3.000 ms  | 4.000 ms  |
```

Fig. 10. Task and memory test result

The use of Memory<T> allows for the safe sharing of data between different threads, without creating redundant copies. Processing with Memory<T> will be faster because Memory<T> reduces the cost of creating new copies of the data and provides more efficient access to memory.



```
0 references
public async Task ParallelProcessingWithString()
{
    var tasks = new Task[_blockCount];

    for (int blockIndex = 0; blockIndex < _blockCount; blockIndex++)
    {
        int start = blockIndex * _blockSize;
        int end = start + _blockSize;
        string blockString = _largeText.Substring(start, _blockSize);

        tasks[blockIndex] = Task.Run(() =>
        {
            ProcessTextBlock(blockString);
        });
    }

    await Task.WhenAll(tasks);
}
```

Fig. 11. Parallel processing with string

III. Results

The results of comparing different memory management techniques in C# - ReadOnlyMemory<T>, ReadOnlySpan<T>, Memory<T>, and Span<T> - demonstrates clear advantages in performance optimization for HTTP request handling. Key findings are as follows:

- ReadOnlyMemory<T>: Offers efficient read-only data access with reduced memory allocations and garbage collection pressure. It is well-suited for handling immutable HTTP request data like headers and bodies, providing safety with minimal performance overhead.

- ReadOnlySpan<T>: Provides the most efficient, zero-allocation solution by leveraging stack-based memory management. It excels in short-lived, high-performance scenarios, such as parsing HTTP requests without persisting data. However, it is limited by its inability to be used across asynchronous boundaries.

- Memory<T>: Allows mutable access to memory, making it versatile for scenarios where data needs to be modified. Its ability to reuse memory buffers reduces garbage collection events and enhances performance in systems with high-throughput HTTP requests.

- Span<T>: Similar to Memory<T>, but focused on stack-based memory, making it ideal for fast, non-persistent data processing. It shares the limitations of ReadOnlySpan<T> in async contexts but provides excellent performance when dealing with mutable, transient data.

Overall, these advanced memory manipulation types significantly reduce memory allocations, improve execution time, and lower GC pressure compared to traditional approaches like StreamReader. The most substantial performance gains are observed when using Span<T> and ReadOnlySpan<T>, which minimize overhead through zero-copy, stack-based memory operations.

IV. Discussion

The special language features in C# and other modern languages that allow more efficient management of sequentially located data structures in memory are relatively new, and there is not much scientific research related to them.

The study [11] is focused on how Span is designed to optimize memory usage and enhance processing speed in .NET applications, with an emphasis on the characteristic collections, which generally store the data in the heap memory, which consumes more RAM and increases the workload of the Garbage Collector. One of the strongest features of Span is to keep data on the stack which enables better performance.

The study [12] presents an in-depth comparison of the use of different algorithms on the traditional List, Array, etc. collections. Experimentation and analysis reveal the different performance of these algorithms using C#.

A comparison of the effectiveness of different strategies related to the optimization of memory management procedures and in particular the release of resources implemented in the C#, Java and C++ languages is discussed in [13]. The surprising conclusion is made that C#'s garbage collection system consistently outperformed the others due to its optimized procedures of asynchronously deallocating memory.

Notorious key techniques in Memory Optimization include avoiding unnecessary object allocations; use value types for small, immutable data; pool reusable objects with ObjectPool<T>; optimize collections (e.g. prefer array or Span<T> over List<T> when possible); avoid large object heap fragmentation by reusing buffers; utilize asynchronous programming effectively to reduce memory pressure.

The test results clearly demonstrate that leveraging modern memory types in C#, such as ReadOnlyMemory<T>, ReadOnlySpan<T>, Memory<T>, and Span<T>, provides significant performance improvements in scenarios involving HTTP request handling. The implications of these findings are particularly relevant for applications that deal with high volumes of HTTP traffic, where memory allocations and GC overhead can become major bottlenecks.

D. Memory Efficiency and Allocation Reduction

One of the key benefits of using these memory types is the reduction in memory allocations. Traditional methods, such as reading data with StreamReader, create new strings and objects on the heap, which leads to frequent memory allocations. This not only consumes more memory but also increases the frequency of GC cycles, leading to performance degradation.

By contrast, Memory<T> and Span<T> significantly reduce the need for heap allocations by reusing memory buffers or leveraging stack-based memory management. This is particularly important in real-time systems or services that handle numerous HTTP requests, as it helps maintain consistent performance under heavy load. The zero-allocation nature of ReadOnlySpan<T> and Span<T>, in particular, shows tremendous potential for short-lived operations where both memory and speed are critical.

E. Garbage Collection Optimization

Reducing GC pressure is a critical factor in achieving high-performance applications, especially in scenarios involving concurrent HTTP requests. The frequent creation and destruction of objects in heap-based memory can result in excessive GC activity, leading to increased latency and jitter. The findings highlight that Memory<T> and Span<T>, by reducing object creation, lead to fewer GC interruptions and smoother application performance.

In particular, ReadOnlyMemory<T> and ReadOnlySpan<T> proved highly efficient for handling immutable data like HTTP headers or request bodies, where copying or modifying data is unnecessary. These types allow direct access to memory without the need for costly allocations, which lowers GC frequency and minimizes its impact on performance.

F. Trade-offs and Limitations

While these memory types offer clear performance gains, there are trade-offs that developers need to consider. For instance, Span<T> and ReadOnlySpan<T> are stack-allocated and, therefore, cannot be used across asynchronous method calls. This limits their applicability in scenarios where asynchronous programming is heavily used, such as modern HTTP request pipelines built on async/await patterns. In these cases, developers must rely on heap-based Memory<T> or ReadOnlyMemory<T>, which still provide performance benefits but with slightly higher overhead compared to their stack-based counterparts.

Additionally, while these types reduce memory allocations and improve performance, they introduce added complexity in managing memory manually. Developers must be more mindful of buffer management and ensure that memory is handled properly to avoid issues such as memory leaks or unsafe access. This represents a trade-off between performance and ease of use, particularly for teams or projects where rapid development is prioritized over fine-tuned optimization.

G. Practical Implications for HTTP Request Handling

For real-world applications that handle HTTP requests, especially high-throughput services such as web servers, API gateways, or microservices, the use of Memory<T> and Span<T> can lead to substantial performance improvements. The ability to avoid data copying, minimize memory fragmentation, and reduce GC pressure can help these systems scale more effectively, handling larger workloads with lower resource consumption.

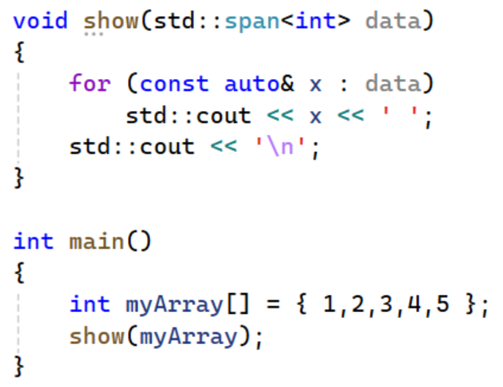
However, these performance benefits come with the requirement for a deeper understanding of memory management in .NET. Developers will need to weigh the trade-offs between the additional complexity and the performance gains, particularly when deciding between the simplicity of StreamReader and the efficiency of the newer memory types.

H. Future Considerations

As C# continues to evolve, further optimizations and tools that will make memory management both easier and more efficient can be expected. The results suggest that adopting these newer memory types now can provide immediate benefits in terms of performance, but future improvements in language features, runtime optimizations, and libraries may help bridge the gap between ease of use and high performance. Additionally, as more developers adopt these patterns, best practices will likely emerge, helping to mitigate the challenges associated with the manual memory management required by Span<T> and Memory<T>.

I. Similar Features in Other Programming Languages

Languages like C++, Rust, D, and Swift have similar features. C++ has std::span which is a very lightweight abstraction (a class template), but powerful tool for working with contiguous sequence of data. A typical implementation holds a pointer to the data, if the extent is dynamic, the implementation also holds a size. The main advantage is that it is a non-owning type (a reference-type rather than a value type). It never allocates nor deallocates anything and does not keep smart pointers alive.



```
void show(std::span<int> data)
{
    for (const auto& x : data)
        std::cout << x << ' ';
    std::cout << '\n';
}

int main()
{
    int myArray[] = { 1,2,3,4,5 };
    show(myArray);
}
```

Fig. 12. Using std::span as function parameter.

In C++, std::span can also be used to simplify syntax. For example, the implementation of arrays in C++ is such that they "don't know" their size. When an array is passed as an argument to a function, actually a constant pointer to the first element of the array is passed and the function "does not know" the number of elements in the array. However, if this array is passed to the function as span, there is no need to pass an additional parameter with the size of the array (Fig. 12).

Instead of Span, Rust has a slice, which is a view into a collection of elements, like std::span. D language offers slices as well, providing a safe way to handle arrays. Swift has ArraySlice, which allows sub-ranges of an array without copying. Each of these languages emphasizes safety and efficiency in handling contiguous data sequences.

Java doesn't have a direct equivalent of Span, but List.subList() allows to work with a sublist view of a list. Although it doesn't offer the same level of low-level control, it does provide a way to handle segments of collections efficiently.

V. Conclusion

Both Span<T> and Memory<T> provide powerful ways to work with memory in C#, especially when performance and efficiency are critical. Overall, our findings confirm that using ReadOnlyMemory<T>, ReadOnlySpan<T>, Memory<T>, and Span<T> provide significant improvements in performance, particularly for HTTP request processing. While there are trade-offs in terms of complexity and applicability in asynchronous programming, these memory types offer a powerful way to optimize memory usage, reduce GC overhead, and improve the scalability of modern .NET applications.

Developers aiming for high-performance HTTP request handling should seriously consider integrating these memory types into their systems to achieve better throughput and responsiveness.

Other practical applications of Span<T> include image processing or numerical computations – situations when working with performance-critical code. In this case, Span<T> can help avoid allocations and reduce the overhead of garbage collection. Similar is the situation when interaction with native code via P/Invoke or other interop mechanisms is needed. Here Span<T> can be used to represent contiguous memory regions efficiently.

Wherever working with asynchronous I/O operations, data buffers or lazy initialization are needed, Memory<T> is a great solution.

References

[1] Microsoft .NET documentation, Microsoft Learn, Span<T> Struct, online: https://learn.microsoft.com/en-us/dotnet/api/system.span-1?view=net-9.0

[2] Microsoft .NET documentation, Microsoft Learn, Memory<T> Struct, online: https://learn.microsoft.com/en-us/dotnet/api/system.memory-1?view=net-9.0

[3] Smith, J., Build your own web server from scratch in Node.JS: Learn network programming, HTTP, and WebSocket by coding a web server (Build Your Own X From Scratch), Independently published, 2024

[4] Jones, R., Hosking, A., Moss, E., The garbage collection handbook: The art of automatic memory management, Chapman and Hall/CRC; 2nd edition, 2023

[5] Price, M., C# 9 and .NET 5 – Modern cross-platform development: Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code, Packt Publishing; 5th edition, 2020

[6] Albahari, J., C# 12 in a Nutshell: The Definitive Reference, O'Reilly Media; 1st edition, 2023

[7] Kokosa, K., Nasarre, Chr., Gosse, K., Pro .NET memory management: For better code, performance, and scalability, Apress; Second edition, 2024

[8] Rasmussen, B. (2014), High-performance Windows Store apps (developer reference), Microsoft Press; 1st edition, 2014

[9] Cormen, T., Leiserson, Ch., Rivest, R., Stein, C., Introduction to Algorithms, The MIT Press; 4th edition, 2022

[10] Sarcar, V., Parallel programming with C# and .NET: Fundamentals of concurrency and asynchrony behind fast-paced applications, Apress, 2024

[11] Akdoğan, H., Duymaz, H., Kocakır, N., Karademir, Ö., Performance analysis of Span data type in C# programming language. Turkish Journal of Nature and Science. October 2024; Issue 1, pp. 29-36. doi:10.46810/tdfd.1425662

[12] Shastri, S., Singh, A., Mohan, B., Mansotra, V., Run-time analysis of searching and hashing algorithms with C#, 2016, Available from: https://www.researchgate.net/publication/326331475_Run-Time_Analysis_of_Searching_and_Hashing_Algorithms_with_C

[13] Henriques, L., Bernardino, J., Performance of memory deallocation in C++, C# and Java, CAPSI 2018 Proceedings. 10., https://aisel.aisnet.org/capsi2018/10