Volume Special Issue, Issue 1, Page 29-36, 2024 https://doi.org/10.46810/tdfd.1425662 Research Article

BİNGÖL UNIVERSITY
Türk Doğa ve Fen Dergisi
Turkish Journal of Nature and Science
www.dergipark.gov.tr/tdfd

Performance Analysis of Span Data Type in C# Programming Language

Hakan AKDOĞAN[1*], Halil İbrahim DUYMAZ[1], Nadir KOCAKIR[1], Önder KARADEMİR[1]

[1*] Özdilek Ev Tekstil San. ve Tic. AŞ, Özveri Ar-Ge Merkezi, Bursa, Türkiye
Hakan AKDOĞAN ORCID No: 0009-0000-5067-268X
Halil İbrahim DUYMAZ ORCID No: 0009-0005-8406-8831
Nadir KOCAKIR ORCID No: 0000-0001-7421-0631
Önder KARADEMİR ORCID No: 0000-0001-5757-7335

**Corresponding author: hakan.akdogan@ozdilek.com.tr*

(Received: 25.01.2024, Accepted: 07.03.2024, Online Publication: 01.10.2024)

Keywords
C#,
Span Data Type,
Performance Analysis,
.NET Core

Abstract: This study presents a comparative analysis of the Span data type in the C# programming language against other data types. Span is a data type supported in .NET Core 2.1 and later versions, and this research investigates its impact on method performance and memory usage. The primary objective of the study is to highlight the potential advantages of the Span data type for C# developers. In pursuit of this goal, the study examines the performance effects of the Span data type through comparative analyses using various methods. For instance, when comparing the StringReplace and SpanReplace methods over 1000 iterations, it is observed that SpanReplace is significantly faster. Similarly, analyses conducted on methods like Contains used in data types such as Queue, List, and Stack demonstrate the performance advantages of the Span data type. In scenarios where the Span data type is employed, it is determined that memory consumption is lower compared to other data types. These findings can assist C# programmers in understanding the potential of the Span data type and optimizing their code accordingly. The Span data type may be a more effective option, especially in data processing and performance-sensitive applications.

C# Programlama Dilinde Span Veri Türünün Performans Analizi

Anahtar Kelimeler
C#,
Span Veri Tipi,
Performans Analizi,
.NET Core

Öz: Bu çalışma, C# programlama dilindeki Span veri tipinin diğer veri tipleriyle karşılaştırmalı analizini sunmaktadır. Span, .NET Core 2.1 ve sonrasında desteklenen bir veri tipidir ve bu araştırma, bu veri tipinin metot performansı ve bellek kullanımı üzerindeki etkilerini araştırmaktadır. Çalışmanın temel amacı, C# geliştiricilerine Span veri tipinin potansiyel avantajlarını vurgulamaktır. Bu amaç doğrultusunda, çeşitli metotlar kullanılarak yapılan karşılaştırmalı analizlerle Span veri tipinin performans üzerindeki etkileri incelenmiştir. Örneğin, StringReplace ile SpanReplace metotları 1000 iterasyonda karşılaştırıldığında, SpanReplace'in önemli ölçüde daha hızlı olduğu görülmüştür. Benzer şekilde, Queue, List, Stack gibi veri tiplerinde kullanılan Contains metodu üzerinde yapılan analizler de Span veri tipinin performans avantajlarını göstermiştir. Span veri tipinin kullanıldığı senaryolarda, bellek tüketiminin diğer veri tiplerine göre daha düşük olduğu belirlenmiştir. Bu bulgular, C# programcıları için Span veri tipinin potansiyelini anlamalarına ve kodlarını optimize etmelerine yardımcı olabilir. Span veri tipi, özellikle veri işleme ve performans hassas uygulamalarda daha etkili bir seçenek olabilir.

1. INTRODUCTION

The ever-evolving landscape of programming languages demands a continual exploration of novel features and data types to enhance code efficiency and application performance. In this context, the present study conducts a comprehensive comparative analysis focusing on the Span data type within the C# programming language, contrasting its attributes against other prevalent data types. Span, introduced and supported in .NET Core 2.1 and subsequent versions, emerges as a key subject of investigation. This research delves into the intricate dynamics of the Span data type, aiming to discern its influence on method performance and memory utilization.

The primary objective of this study is to elucidate the latent advantages that the Span data type offers to developers immersed in the C# programming paradigm. To achieve this goal, the research employs a systematic approach, scrutinizing the performance implications of the Span data type through meticulous comparative analyses leveraging various methods.
Furthermore, the study extends its inquiry to encompass analyses of methods such as Contains, Slice, SubString, StartsWith and Replace commonly employed in data types like String, Queue, List, and Stack. The outcomes of these analyses consistently underscore the performance advantages inherent in the utilization of the Span data type. Notably, in scenarios where the Span data type finds application, discernible reductions in memory consumption are identified in contrast to other data types.

The findings of this investigation bear substantial implications for C# programmers, providing valuable insights into the untapped potential of the Span data type. Armed with this knowledge, developers can make informed decisions in optimizing their code to harness the advantages offered by the Span data type. Consequently, the Span data type emerges as a promising and more effective option, particularly in domains that demand enhanced data processing capabilities and cater to performance-sensitive applications.

2. LITERATURE REVIEW

Code effiency and application performance has been a topic of growing interest in the field of software engineering. This literature review aims to provide a comprehensive overview of existing studies, methodologies, and advancements.

The study titled "Performance Characterization of .NET Benchmarks," conducted by Deshmukh et al. [1], published in IEEE in 2021, investigates hardware performance bottlenecks in .NET applications. Employing Principal Component Analysis (PCA) and hierarchical clustering on open-source .NET and ASP.NET benchmarks, the research reveals that these applications possess distinct characteristics compared to traditional SPEC-like programs. Consequently, this dissimilarity underscores the need for consideration in architectural research. The study highlights that .NET benchmarks exhibit a significantly higher front-end dependency and analyzes the effects of managed runtime events, such as Garbage Collection (GC) and Just-in-Time (JIT) compilation.

The article titled Measuring Performance Improvements in .NET Core with BenchmarkDotNet by Almada [2], investigates the performance implications of value-type versus reference-type enumerators in C#. The author begins by highlighting how the C# compiler generates different code for the 'foreach' keyword based on the type of the collection. The critical distinction lies in whether the GetEnumerator() method returns a value type or a reference type enumerator, which significantly impacts collection iteration performance. Reference-type enumerators, associated with classes and interfaces, involve virtual calls and heap allocations, potentially affecting performance. On the other hand, value-type enumerators, exemplified by collections like List<T>, demonstrate improved performance, especially for large collections, and avoid heap allocations. The article includes benchmarking using BenchmarkDotNet, comparing the performance of iterating a List<int> when cast to IEnumerable<int> (reference-type enumerator) versus using List<int> directly (value-type enumerator).

The results indicate substantial performance differences, with value-type enumerators outperforming their reference-type counterparts. The article concludes by emphasizing the importance of considering the implications of virtual calls, recommending the use of value-type enumerators for better performance in collection iteration, and encouraging the adoption of immutable collections to avoid casting to interfaces. The benchmarks conducted on various .NET versions highlight performance improvements, providing additional motivation for transitioning to the latest versions. This insight contributes valuable considerations for developers seeking to optimize collection iteration performance in C#.

The research paper by Usman et al. [3], the authors conduct a performance analysis of searching algorithms in C#. The study focuses on evaluating the efficiency of various searching algorithms, including linear search, binary search, and brute force search, measured in terms of time complexity. The algorithms are implemented in the C family of compilers, including C#, and their performance is assessed on different machines using a sample file. The analysis considers the execution time of searching algorithms, with variations observed based on different systems and file sizes.

The results suggest that linear search exhibits better time complexity, while brute force search excels in finding all search patterns. The paper emphasizes the significance of efficient searching in programming, highlighting its importance for system throughput and efficiency. Experimental works include the development of a C# program incorporating the mentioned algorithms and a detailed examination of their performance on machines with varying processing power. The authors conclude by discussing the implications of their findings and propose future work involving the implementation of these techniques on other compilers and exploring space complexity considerations.

Shastri et al. [4], offer a comprehensive examination of various searching and hashing algorithms, shedding light on their efficiencies in terms of time complexity. The study delves into five distinct algorithms, namely Linear Search, Binary Search, Interpolation Search, Division Method Hashing, and Mid Square Method Hashing. Through rigorous experimentation and analysis, the authors demonstrate the performance of these algorithms using Visual Studio C#. They meticulously evaluate the run-time of each algorithm across different input sizes, ranging from 10,000 to 100,000 elements, and highlight the advantages and disadvantages of each. Notably, the findings reveal that Binary Search consistently outperforms other algorithms, showcasing its superiority, particularly for large datasets. This insightful exploration contributes valuable insights to the field of algorithm analysis and aids in understanding the optimal choices for various search and hash functions in real-world applications.

Arif et al. [5], provide an empirical analysis of C#, PHP, JAVA, JSP, and ASP.Net with a focus on performance analysis based on CPU utilization. The paper underscores the significance of software development within the context of computer systems, emphasizing the need for enhanced speed and reliability in modern digital systems. Through extensive research, the authors explore the impact of different programming languages on computer performance and resource utilization, with a particular emphasis on CPU usage. Their investigation involves the development of a software application using these languages, focusing on factors such as algorithm quality, programming language selection, and SQL query optimization. The findings suggest that JAVA/JSP exhibit superior performance compared to other languages in terms of CPU usage, memory utilization, and execution time, providing valuable insights for developers and software architects aiming to optimize system performance.

Sestoft [6], explores the comparative numeric performance of C, C#, and Java across various small-scale computational tasks. While managed languages like C# and Java offer ease of use and safety, their performance in numeric computations, especially involving arrays or matrices of floating-point numbers, is comparatively inferior to that of traditional languages like C and C++. This performance gap arises due to differences in compiler optimization strategies, array access overheads, and the need for index checks in managed languages, which can incur additional execution overhead and hardware slowdowns. However, the study demonstrates that with careful optimization techniques, such as employing unsafe code in C# or leveraging high-performance virtual machines in Java, it's possible to narrow this performance gap, showcasing the nuances and trade-offs in numeric computation across these languages.

This study presents a comparative analysis of the Span data type, supported in .NET Core 2.1 and later versions, within the context of other data types in the C# programming language. While existing literature has analyzed various aspects of .NET applications, including hardware performance bottlenecks, C# collection iteration performance, the efficiency of different search algorithms, and the impact of programming languages on CPU usage, this work specifically focuses on the effects of the Span data type on method performance and memory utilization, adding a new dimension to the discourse. Previous studies have primarily concentrated on performance comparisons of specific algorithms, programming languages, or .NET versions. In contrast, this study examines the direct impacts of utilizing the Span data type in terms of method performance and memory usage, aiming to highlight its potential advantages for C# developers. By comparatively analyzing the performance differences in specific methods such as StringReplace and SpanReplace, and the effects on memory consumption in data types (Queue, List, Stack) when methods like Contains are used, this work demonstrates that the Span data type can be a more effective option for applications with high performance sensitivity and data processing requirements. Therefore, this study significantly contributes to the existing literature by revealing the potential that the Span data type, introduced in newer versions of .NET Core, offers for code optimization within the C# programming paradigm, using academic language.

3. MATERIAL AND METHOD

3.1. Span Data Type

The methodology employed in this research revolves around a comprehensive investigation of the Span data type in C#, introduced in version 7.2 and supported in .NET Core 2.1 and subsequent releases. The primary objective of the Span data type is to efficiently handle data in memory and expedite processing procedures, making it an ideal choice for operations involving substantial datasets.
The robust aspects and limitations of the Span data type are categorized under specific headings, facilitating a more straightforward interpretation of the data collected in the study.

3.1.1. Memory management and performance

The study is focused on how Span is designed to optimize memory usage and enhance processing speed in .NET applications. Traditional collection structures often maintain data on the heap, leading to unnecessary RAM occupation and increased Garbage Collector workload. Span, however, stores data pointers on the Stack. The Stack, typically limited to around 4 MB per application, provides faster access compared to the heap. Given these considerations, the Stack offers significantly faster access than the heap. Therefore, the ability of Span to keep data on the stack enables swift operations.

In Figure 1, the utilization of stack and heap memory for the String data type is illustrated.

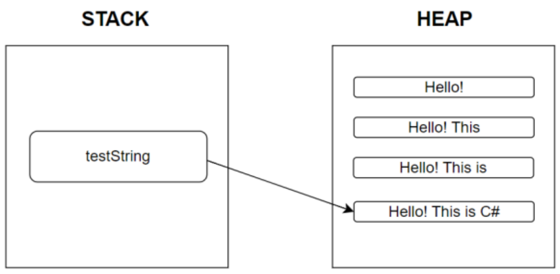

Figure 1. The way string uses stack and heap memories [7]

In Figure 2, the utilization of stack memory by the Span data type is illustrated.

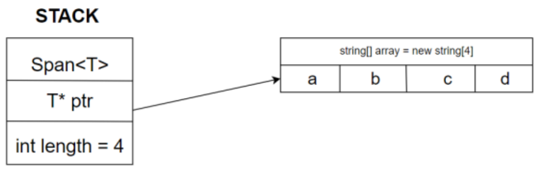

Figure 2. The way span use stack memory [8]

3.1.2. Reduced memory footprint

Span allows the usage of data on the stack without memory allocation by pointing to it. This is particularly advantageous in operations such as data copying, slicing, and sorting. Unlike traditional collections that produce new objects for such operations, Span directly accesses existing data through pointers stored on the stack. This prevents unnecessary memory allocations, contributing to more efficient application performance.

3.1.3. Memory efficiency

The study highlights the significant advantages of Span in terms of memory efficiency. For instance, when working with a string, traditional substring methods create new references in memory for each extracted portion, leading to excessive memory consumption and increased Garbage Collector activity. Span, on the other hand, maintains an offset and length on the stack, avoiding the creation of new references on the heap. This offset and length point to the desired portion of the string data on the heap, allowing operations to be completed without creating new references in memory.

3.1.4. Ref struct

The Span is categorized as a ref struct type, indicating specific limitations. Span cannot operate under any async-marked methods or Iterator methods. This restriction arises from the usage of local variables in both methods. Unlike these methods, Span avoids local variable assignments to preserve data and disposes of data from memory when the scope is exited. Additionally, ref struct types cannot implement interfaces.

3.2. Environment and Libraries

In the conducted study, based on the information gathered about the Span data type, tests were conducted using both simulated data and real data from applications used in our company. These tests were performed using C# and .NET Framework 7.0. In order to conduct controlled iterations for testing purposes and perform data manipulation using predefined functions, the BenchmarkDotNet library was used. The performance difference between the Span data type and other included data types was revealed through these tests. All tests were conducted in the same environment, and multiple repetitions ensured the elimination of undesired conditions that could impact test results.
The physical characteristics of the testing environment and the libraries employed are presented in the below table.

3.2.1. Test environment

In Table 1, certain specifications regarding the physical environment in which the experiments were conducted, including Processor, RAM, Operating System, and the necessary library for performance measurement during the experiments, are provided.

Table 1. Test Environment

Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
RAM	16 GB
Operating System	Microsoft Windows 11 Pro
Library	BenchmarkDotNet-0.13.5

3.2.2. BenchmarkDotNet

BenchmarkDotNet is a versatile tool designed to seamlessly convert methods into benchmarks, monitor their performance, and facilitate the dissemination of reproducible measurement experiments. This library simplifies the benchmarking process, akin to composing unit tests. Its internal mechanisms leverage sophisticated techniques, notably the perfolizer statistical engine, ensuring the generation of reliable and precise results. Beyond its facilitative role, BenchmarkDotNet serves as a protective barrier against common benchmarking pitfalls, offering alerts for potential issues within benchmark design or acquired measurements. The output is presented in an intuitive format that accentuates crucial details about the experimental outcomes [9].

4. DATA ANALYSIS AND DISCUSSION

In this section, an in-depth analysis is presented concerning the performance aspects observed within the context of a online education portal project developed at Özdilek Özveri R&D Center. The examination focuses on empirical evaluations conducted on genuine user data, exceeding a count of 80,000 records, to systematically discern the performance distinctions between the Span and List data types. Special attention is devoted to benchmark tests meticulously devised to highlight the unique performance attributes of the Span data type, particularly in the realm of data manipulation within the String data type. The dataset utilized for these benchmark tests is structured in JSON format, ensuring consistency across each test iteration. The ensuing analysis and discussion shed light on the nuanced aspects of performance exhibited by these data types in real-world applications, providing valuable insights into their respective strengths and capabilities.

The JSON data utilized in real-world data tests is presented in Figure 3 below.

```
{
  "email" : "example@gmail.com",
  "title" : "Software Assistant Specialist",
  "fullName" : "Hakan Akdoğan",
  "isActive" : true,
  "employeeNo" : "00001",
  "positionCode" : "99999",
  "isServiceData" : true,
  "expenseLocationName" : "Bursa",
  "employeeField" : "2000",
  "identityNumber" : "11111111111",
  "workerSubGroup" : "11",
  "employeeSubField" : "2008",
  "employeeSubFieldName" : "R&D",
  "employeeFieldDescription" : "Research And Development"
}
```

Figure 3. JSON User Data

4.1. User Data Benchmark Tests

4.1.1. User data contains method benchmark

Contains Method: The Contains method is used to check if a specified element is present in a collection, such as an array or list. It returns true if the element is found and false if not. This method is commonly used to determine the existence of an item in a dataset, providing a straightforward way to perform such checks in .NET applications.
In this benchmark test, the presence of the term "example" is being examined within the email field of the JSON where user data is stored [10].

As illustrated in the figure below, the Contains method applied on Span exhibited a performance approximately two times faster than when applied on List.

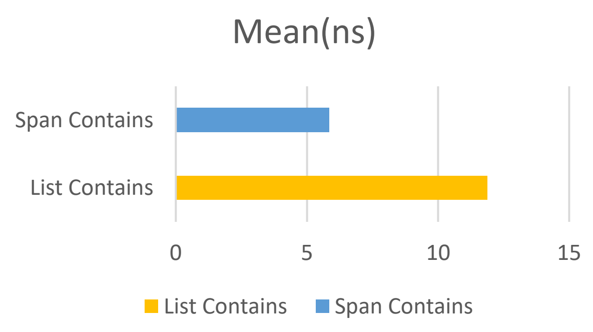

Figure 4. Contains Method Benchmark

4.1.2. User data binary search method benchmark

Binary Search Method: The Binary Search method is employed to efficiently locate a specified value within a sorted collection, such as an array or list. It follows a binary search algorithm, systematically dividing the collection in half and determining whether the sought value lies in the first or second half. This process continues until the exact position of the value is identified or it is confirmed that the value is not present in the collection. The binary search method is particularly advantageous for large datasets due to its logarithmic time complexity, resulting in faster search operations compared to linear search algorithms.
In this benchmark test, the presence of the term "example" is being examined within the email field of the JSON where user data is stored [11].

As illustrated in the figure below, the Binary Search method applied on Span concludes the operation in approximately 55 ns, whereas List completes the same operation in around 73 ns.

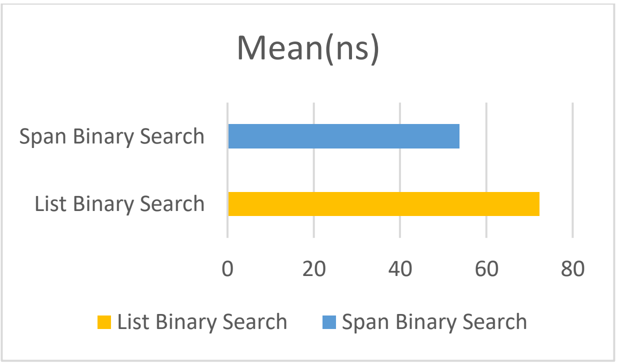

Figure 5. Binary Search Method Benchmark

4.1.3. User data slice method benchmark

Slice Method: The Slice method efficiently extracts a contiguous subset of elements from a data structure, like an array or a Span. Rather than copying elements, it creates a new view or reference to the original data, allowing for enhanced performance and reduced memory usage when working with specific data segments. This feature is particularly useful for handling large datasets with improved efficiency [12].

As illustrated in the figure below, the Slice method applied on Span accomplishes the task of cutting a JSON array in approximately 1 ns, whereas, in contrast, this operation takes around 15 ns when performed on a List.

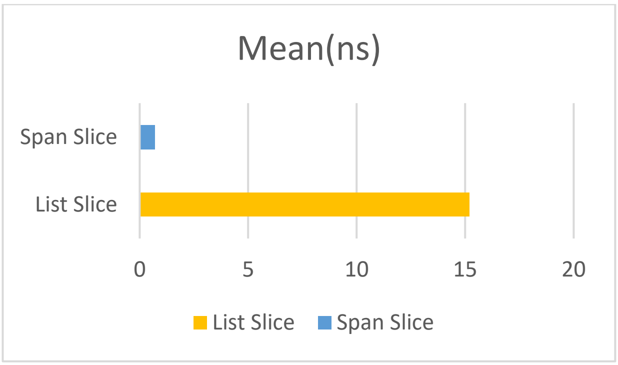

Figure 6. Slice Method

4.2. Example Data Benchmark Tests

In this section, benchmark tests have been conducted by iterating 10,000 times over the sentence 'This is an example sentence.' for performance evaluation.

4.2.1. Example data replace method benchmark

Replace Method: The Replace method is a string manipulation function that replaces all occurrences of a specified substring with another substring within a given string. It provides a straightforward way to modify string content by substituting specified patterns with desired values. The method takes two string parameters: the first represents the substring to be replaced, and the second represents the new substring. The replacement is applied to all instances of the specified substring in the original string. This function is commonly used for simple text transformations and substitutions within string data [13].

In this benchmark test, the Replace method was applied to substitute the substring "an example" with "a sample" in the sentence "This is an example sentence." As illustrated in the figure below, when applied to a Span, the operation concludes in approximately 25,000 ns, while on a String, it takes about 430,000 ns to complete. This comparison highlights the substantial performance advantage of using the Replace method with Span over String.

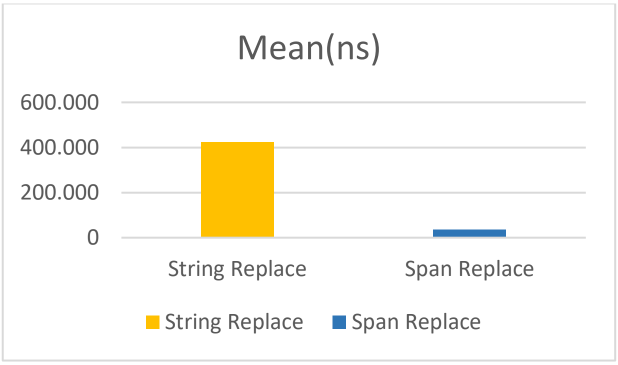

Figure 7. Replace Method

As illustrated in the graph below, when this method is applied to a Span, it performs the operation without any memory allocation, whereas when applied to a String, it allocates approximately 500,000 B of memory. This observation underscores the efficient memory handling of the method when used with Span compared to the memory allocation associated with its application on String.

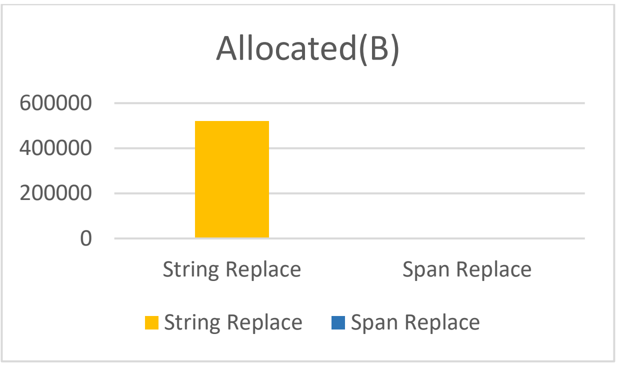

Figure 8. Replace Method Allocated Memory

4.2.2. Example data slice method benchmark

In the benchmark test, the Slice method was applied to truncate the sentence "This is an example sentence" from the first character to the 20th character, resulting in the string "This is an example se". As illustrated in the figure below, the tests conducted with the Slice method revealed that the utilization of Span is approximately 9 times faster than its counterpart using String.

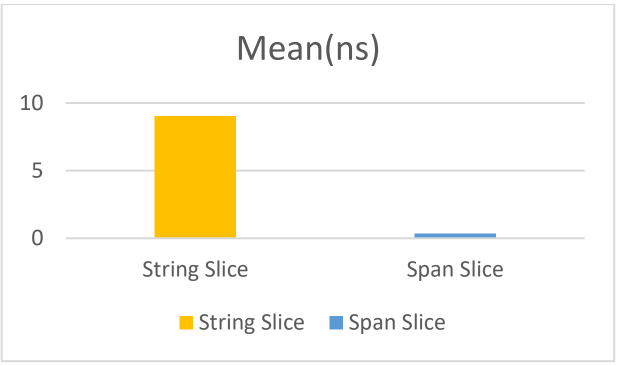

Figure 9. Slice Method

4.2.3. Example data startsWith method benchmark

StartsWith Method: The StartsWith method is used to determine whether a given string begins with a specified prefix. It returns a boolean value indicating whether the string starts with the provided prefix. This method is commonly employed to perform simple prefix-based checks in strings without the need for manual character comparisons. The result is true if the string starts with the specified prefix; otherwise, it returns false. The method is straightforward and provides a convenient way to validate the initial portion of a string in various applications, including text processing and pattern matching [14].

In this benchmark test, the StartsWith method was employed to determine whether the sentence "This is an example sentence" begins with the word "This." As illustrated in the figure below, the comparison using the StartsWith method revealed a significant performance gap. While the String implementation took nearly 38,000 ns to complete this operation, the Span implementation accomplished it in a mere 2.8 ns. This emphasizes the noteworthy efficiency advantage of the Span type in performing prefix-based checks compared to traditional String operations.the benchmark test, the Slice method was applied to truncate the sentence "This is an example sentence" from the first character to the 20th character, resulting in the string "This is an example se". As illustrated in the figure below, the tests conducted with the Slice method revealed that the utilization of Span is approximately 9 times faster than its counterpart using String.

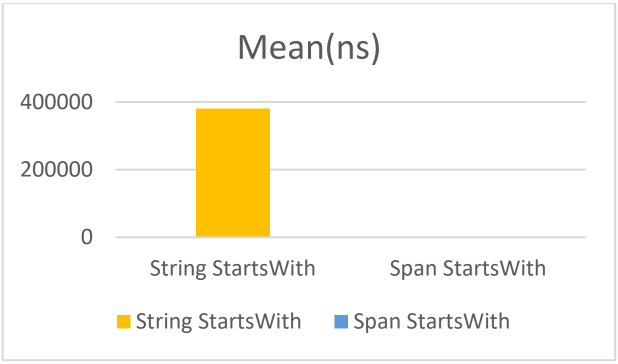

Figure 10. StartsWith Method

4. CONCLUSION

The findings from our comprehensive investigation into the Span data type in C# programming unequivocally demonstrate its superior efficiency in memory utilization and processing speed relative to traditional data types. Our research, grounded in a methodical comparative analysis of real-world and synthetic datasets, highlights the significant advantages of employing Span across various computational methods. Notably, our results reveal that SpanReplace consistently outshines StringReplace in terms of performance over 1000 iterations, thereby affirming the efficiency gains attainable with Span's adoption.

Our examination of memory consumption patterns further underscores the importance of Span, showcasing a marked reduction in memory footprint when juxtaposed with conventional data types. This facet is particularly salient in the processing of large datasets, where efficient memory management is directly linked to the overall performance of applications.
The implications of our findings extend well beyond the realm of performance enhancement. They furnish C# developers with valuable insights into codebase optimization, enabling them to leverage the full potential of the Span data type. With this knowledge, developers are better equipped to tackle the challenges of performance-sensitive applications, benefiting from the tangible improvements in speed and memory efficiency that Span offers.

Looking forward, our study serves as a robust foundation for future research focused on the Span data type. Further investigations could explore specific use cases or optimization strategies across various application domains, enriching our understanding of Span's performance characteristics through the inclusion of empirical data from a broader array of scenarios.
In summary, our research elucidates the transformative potential of the Span data type within C# programming, heralding it as a pivotal choice for high-performance and memory-efficient applications. As the software development landscape evolves, the insights derived from our study will undoubtedly contribute to ongoing advancements in code efficiency and application performance.

Acknowledgement

This study was presented as an oral presentation at the "6th International Conference on Life and Engineering Sciences (ICOLES 2023)" conference.

REFERENCES

[1] Deshmukh R, Li R, Sen RR, Henry M, Beckwith G, Gupta G. Performance characterization of .NET benchmarks. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); 2021 Apr. Stony Brook, NY, USA. p. 107-17. doi: 10.1109/ISPASS51385.2021.00028.
[2] Almada A. Performance of value-type vs. reference-type enumerators in C# [Internet]. 2023 Jul 22 [cited 2023 Jul 22]. Available from: https://www.linkedin.com/pulse/performance-value-type-vs-reference-type-enumerators-ant%C3%A3o-almada/
[3] Usman M, Bajwa Z, Afzal M. Performance analysis of searching algorithms in C#. International Journal for Research in Applied Science & Engineering Technology (IJRASET). 2014;2(4):511-3.
[4] Shastri S, Singh A, Mohan B, Mansotra V. Run-time analysis of searching and hashing algorithms with C# [Internet]. 2016 Dec [cited 2016 Dec]. Available from: https://www.researchgate.net/publication/311541937_Run-Time_Analysis_of_Searching_and_Hashing_Algorithms_with_C#
[5] Arif MA, Hossain MS, Nahar N, Khatun MD. An empirical analysis of C#, PHP, JAVA, JSP and ASP.Net regarding performance analysis based on CPU utilization. Banglavision Research Journal. 2014;14(1):174-88.
[6] Sestoft P. Numeric performance in C, C# and Java [Internet]. 2007 Feb 28 [cited 2007 Feb 28]. Available from: https://www.researchgate.net/publication/228380860_Numeric_performance_in_C_C_and_Java
[7] The Tech Platform. Why and how string is immutable in C# [Internet]. 2021 Oct [cited 2021 Oct]. Available from: https://www.thetechplatform.com/post/why-and-how-string-is-immutable-in-c
[8] Wickramarathna N. An introduction to writing high-performance C# using Span<T> struct [Internet]. 2021 Dec [cited 2021 Dec]. Available from: https://nishanc.medium.com/an-

introduction-to-writing-high-performance-c-using-span-t-struct-b859862a84e4

[9] BenchmarkDotNet. BenchmarkDotNet: Powerful .NET library for benchmarking [Internet]. 2024 Jan [cited 2024 Jan]. Available from: https://github.com/dotnet/BenchmarkDotNet

[10] CodeAcademy. Contains() [Internet]. 2023 Apr [cited 2023 Apr]. Available from: https://www.codecademy.com/resources/docs/c-sharp/strings/contains

[11] Tripathi P. Binary search using C# [Internet]. 2023 Nov [cited 2023 Nov]. Available from: https://www.c-sharpcorner.com/blogs/binary-search-implementation-using-c-sharp1

[12] Microsoft. ArraySegment<T>.Slice Method [Internet]. 2024 Jan [cited 2024 Jan]. Available from: https://learn.microsoft.com/en-us/dotnet/api/system.arraysegment-1.slice?view=net-8.0

[13] Geeks for Geeks. C# | Replace() method [Internet]. 2019 May [cited 2019 May]. Available from: https://www.geeksforgeeks.org/c-sharp-replace-method/

[14] Microsoft. String.StartsWith Method [Internet]. 2024 Jan [cited 2024 Jan]. Available from: https://learn.microsoft.com/en-us/dotnet/api/system.string.startswith?view=net-8.0