PERFORMANCE ANALYSIS OF LOOPS IN THE C# PROGRAMMING LANGUAGE

Miša Blagojević[1]*,
[0009-0000-4394-8422]

Mladen Veinović[2]
[0000-0001-6136-1895]

[1]Student,
Singidunum University,
Belgrade, Serbia

[2]Singidunum University,
Belgrade, Serbia

Abstract:

This paper presents an experimental evaluation of loop performance characteristics. The paper suggests methods for improving performance beyond pre-optimisation levels, with an emphasis on execution efficiency for collections of reference-type objects. Unlike prior studies that predominantly focus on primitive data types, this work addresses a more representative use case for more complex reference types.

An experimental evaluation was carried out using the BenchmarkDotNet NuGet package for Visual Studio to measure loop execution speeds for for, while, and do-while loops while iterating through arrays and generic lists. Two widely adopted optimisation techniques—caching of invariant expressions and loop unrolling—were applied and assessed against non-optimised baseline implementations of the same loops. The experimental setup included datasets of varying sizes, specifically 10,000 and 10,001 elements, to evaluate the impact of remainder handling in loop unrolling.

The results indicate that the optimisation techniques provide performance improvements of up to approximately 10%, depending on the loop construct and underlying data structure. For loops demonstrated the most significant performance increases when iterating over arrays, and while loops achieved competitive results in a couple of scenarios involving generic lists.

These findings enhance the understanding of loop-level performance optimisation in C#, offering actionable insights for designing performance-critical components in DotNet-based applications.

The key contributions of this paper are:

- A comparative experiment on loop constructs for reference type collections in C#.
- A quantitative assessment of manual optimisation strategies, such as caching and loop unrolling.
- An examination of how performance is affected by dataset size and remainder handling.

Keywords:

C# Loop Optimisation, Loop Unrolling, Caching Techniques, Benchmarking, C# Performance.

INTRODUCTION

Performance optimisation in managed programming environments, particularly within the .NET framework, remains an important area of both research and practical application.

Actual runtime performance heavily depends on code structure, data access patterns, and developer-applied optimisation techniques. Loop constructs are one of the most frequently executed and optimised components

Correspondence:
Miša Blagojević

e-mail:
misa.blagojevic.25@singimail.rs

DOI: 10.15308/Sinteza-2026-322-328

in software systems, especially in performance-critical applications. Measurable overall gains can be achieved by small optimisations in loop constructs.

Most frequent optimisation techniques, such as loop unrolling and loop caching, have been widely studied in the context of low-level and compiler design. However, less explored are their practical impact within managed environments like C#-particularly when operating over collections of reference-type objects.

Existing studies primarily focus on primitive data types and low-level optimisations, often overlooking real-world scenarios involving more complex data structures such as objects stored in arrays and generic collections. Furthermore, the interaction between manual optimisation techniques and runtime optimisations performed by the JIT compiler requires empirical validation because it is not always straightforward.

To address this gap, this paper provides an experimental evaluation of loop performance in C#, focusing on collections of reference-type objects. The paper compares different loop constructs and evaluates, under controlled conditions, the impact of selected manual optimisation techniques.

The main contributions of this paper include:

- A comparative analysis of the performance of for, while, and do-while loops operating on arrays and generic lists of reference-type objects.
- An experimental evaluation of two widely used optimisation techniques: invariant expression caching and loop unrolling.
- An examination of performance sensitivity to data-set size and remainder handling in loop unrolling.
- Practical guidance for developers seeking to optimise performance-critical code in .NET applications.

2. RELATED WORK

Performance optimisation in managed programming environments, particularly within the .NET framework, has proven to be a rich area for research. Scholars have explored execution efficiency, memory management, and compiler behaviour extensively. Early contributors, such as Albahari [1], explored the complexities of just-in-time (JIT) compilation and its impact on performance.

In the realm of loop optimisation, techniques such as loop unrolling and invariant code motion serve as fundamental compiler-level enhancements. [2] [3] While modern JIT compilers employ various auto-optimisations, their effectiveness largely depends on code structure and runtime context.

Warren [4] highlights low-level optimisation strategies, emphasising the importance of streamlining branching and reducing memory access challenges within loops.

Within the C# and .NET landscape, numerous studies and practical guides examine the delicate balance between iteration constructs and data structures. Skeet [5], along with Troelsen and Japikse [6] provides a detailed analysis of language features and their performance implications. The choice between iterating over arrays and generic collections can lead to different performance outcomes, influenced by memory layout and bounds checking.

The BenchmarkDotNet NuGet package for Visual Studio, highlighted by its developers [7], has become a standard for microbenchmarking within .NET, facilitating precise measurement of subtle performance variations.

Recent studies have examined the costs of high-level abstractions like LINQ and their effects on performance-sensitive paths. [8] Additionally, investigations into boxing and unboxing overhead [9] and memory allocation patterns [10] emphasises the need to minimise runtime overhead in loops.

Despite the extensive research on performance optimisation, there are relatively few studies that focus specifically on loop behaviour concerning collections of reference-type objects in C#. This paper aims to address that gap by providing an empirical comparison of loop constructs and targeted optimisation techniques in controlled environments.

3. LOOP OPTIMISATION

Various strategies can be employed within loops to improve their execution speed. Before implementing optimisation techniques, it is crucial to identify areas for enhancement through benchmarking. This paper will be focused on two of them:

- Loop unrolling
- Loop caching

3.1. LOOP UNROLLING

Loop unrolling is a technique that can enhance program execution speed by decreasing the number of iterations through a loop, although this may lead to an increase in binary size. [11] It can be performed manually or by using an optimising compiler, such as the C# Just-In-Time compiler, which partially implements this technique. The program's speed improves by minimising iterations and reducing branching, among other factors.

When employing loop unrolling, "clean-up" code may be required if the number of iterations is not known at compile time. However, if the number of iterations is known and is a multiple of the unrolling factor, "clean-up" code is unnecessary. The unrolling factor refers to how many times the loop body is executed until the loop is fully unrolled or the iteration space is reduced. If only the iteration space is reduced, then "clean-up" code becomes essential. See Listing 1.

3.2. LOOP CACHING (MANUAL OPTIMISATION)

If it is possible to lighten the load for the loop, every value that does not change, such as the length of the array, collections, and so on, should be saved in a variable outside of the loop, and not evaluated on every iteration. [12] It is called manual optimisation because the developer must write the code.

The following example demonstrates when there is no caching. See Listing 2.

The next example demonstrates when there is caching. See Listing 3.

4. METHODOLOGY

The study was performed on a system running Windows 11 (version 22H2), equipped with an AMD Ryzen 9 5950 processor and 128 GB of RAM. All experiments were conducted in Release mode, using the default .NET runtime and RyuJIT compiler to ensure accurate performance measurements. Benchmarking was performed with the use of a widely adopted tool for microbenchmarking in the dot NET ecosystem, named BenchmarkDotNet.

```
int GetSums()
{
    int sumOfItems = 0;
    int stepNo = 3;
    int numberOfRestOfItems = listOfValues.Length % stepNo;
    int maxNumberOfCompleteIterations = listOfValues.Length - numberOfRestOfItems;
    /*Iterations by chunks of 3 elements*/
    for (int i = 0; i < maxNumberOfCompleteIterations; i = i + stepNo)
    {
        sumOfItems = sumOfItems + listOfValues[i];
 sumOfItems = sumOfItems + listOfValues[i + 1];
 sumOfItems = sumOfItems + listOfValues[i + 2];
    }
    for (int i = maxNumberOfCompleteIterations; i < listOfValues.Length; i++)
    {
        sumOfItems = sumOfItems + listOfValues[i];
    }
    return sumOfItems;
}
```

Listing 1. Loop unrolling

```
for (int i = 0; i < someList.Length; i++)
{
    /*some code*/
}
```

Listing 2. Loop without cashing

```
int arrayLength = someList.Length;

for (int i = 0; i < arrayLength; i++)
{
    /*some code*/
}
```

Listing 3. Loop with cashing

Reliable and reproducible results were ensured because BenchmarkDotNet provides high-precision measurements by automatically handling warm-up phases, multiple iterations, statistical analysis, and outlier removal.

The experiments involved datasets of 10,000 elements, along with an additional dataset of 10,001 elements to evaluate the effects of remainder handling in loop unrolling. Each dataset included objects with one string field and one integer field, representing a simplified model of reference-type collections typically used in practice. While this dataset design allows for controlled and consistent measurements, it should be acknowledged that it does not fully represent all aspects of real-world workloads, especially those involving complex object graphs or memory access patterns.

The following loop constructs were assessed:

- for loop
- while loop
- do-while loop.

They were tested with two data structures:

- Arrays
- Generic lists.

Two optimisation techniques were applied:

- Invariant expression caching eliminates repetitive computations by evaluating loop-invariant values outside the loop.
- Loop unrolling streamlines processes by handling multiple elements in each iteration.

Each benchmark scenario was executed several times to calculate average execution times. Performance improvements were measured as percentage differences from the baseline of non-optimised implementations.

This methodology aimed to highlight the effects of loop structures and optimisation techniques while reducing the impact of external factors, such as hardware variability.

5. OBSERVATIONS

The experimental results highlight several key patterns in loop performance within C#, especially regarding collections of reference-type objects. While raw execution times offer a general view of performance differences, a more detailed analysis reveals the factors influencing these outcomes.

In the baseline scenario, the differences between loop constructs are minor, suggesting that modern JIT compilers effectively optimise standard iteration patterns. See Table 1. However, some variations are still noticeable. The for loop performs best when iterating over arrays, thanks to its straightforward and predictable structure, which allows the JIT compiler to optimise bounds checking and loop control efficiently. Arrays also enhance cache locality due to their contiguous memory layout, enabling quicker sequential access. Conversely, when iterating over generic lists, the do-while loop shows slightly better performance.

Table 1. Results for executing the loop without optimisation applied

Method description	First run	Second run	Third run	Average
FOR_ARRAY	27.11	27.21	27.28	27.20
WHILE_DO_ARRAY	26.90	27.03	27.00	26.97
DO_WHILE_ARRAY	27.21	27.00	27.13	27.11
FOR_GEN_LIST	31.99	32.10	31.86	31.98
WHILE_DO_GEN_LIST	32.12	32.15	32.03	32.10
DO_WHILE_GEN_LIST	30.76	30.73	30.75	30.74

Table 2. Results for executing loops after caching is applied

Method description	First run	Second run	Third run	Average
FOR_ARRAY	27.11	27.21	27.28	27.20
WHILE_DO_ARRAY	26.90	27.03	27.00	26.97
DO_WHILE_ARRAY	27.21	27.00	27.13	27.11
FOR_GEN_LIST	31.99	32.10	31.86	31.98
WHILE_DO_GEN_LIST	32.12	32.15	32.03	32.10
DO_WHILE_GEN_LIST	30.76	30.73	30.75	30.74

This may stem from subtle differences in branching behaviour and loop entry conditions, as well as the internal workings of list access methods, which add more abstraction than arrays.

Introducing caching (Table 2) leads to consistent performance gains across all loop constructs and data structures, with improvements ranging from about 2% up to 10%. These gains are mainly due to the reduction of repeated evaluations of invariant expressions, like collection length. While modern JIT compilers can perform loop-invariant code motion automatically, this optimisation isn't always guaranteed, especially in more complex scenarios or with abstractions involved. The most notable improvements occur in for loops over arrays, indicating that manual caching reduces the overhead in loop condition evaluation, allowing the JIT to produce more efficient machine code. In contrast, the smaller gains in do-while loops over generic lists suggest that other factors, such as method call overhead or memory access patterns, might have a greater impact on performance in those instances.

When loop unrolling is combined with caching (Table 3 and Table 4), additional performance improvements are observed, although these gains are not consistent across the board. Loop unrolling decreases the number of iterations and conditional branch evaluations, which can enhance instruction-level parallelism and reduce branch misprediction penalties. This effect is particularly evident in array-based iterations, where memory access is highly predictable and benefits more from reduced control flow overhead. However, the gains from unrolling are somewhat limited compared to caching alone, suggesting that the JIT compiler may already perform certain low-level optimisations, diminishing the added value of manual unrolling. Moreover, the increased code size from unrolling may negatively affect instruction cache efficiency, partially offsetting performance improvements.

Introducing datasets with 10,001 elements (Table 4) allows for the assessment of clean-up code in loop unrolling scenarios. The results indicate that the presence of remainder handling does not significantly impact performance. This suggests that the overhead from the additional loop is minimal compared to the overall execution time. However, it also points out that the benefits of loop unrolling are sensitive to the characteristics of the dataset, especially when iteration counts do not align perfectly with the unrolling factor.

5.1. ARRAYS VS. GENERIC LISTS

A consistent trend across all experiments is the superior performance of arrays compared to generic lists. This difference can be attributed to several factors:

- Arrays allow direct memory access with minimal abstraction.
- Generic lists necessitate additional bounds checking and method calls.
- The memory layout of arrays is more cache-friendly.

Table 3. Results after caching and loop unrolling, without clean-up code

Method description	First run	Second run	Third run	Average	Percentage
FOR_ARRAY	24.85	24.70	24.59	24.71	+9.15%
WHILE_DO_ARRAY	24.96	24.89	24.83	24.89	+7.17%
DO_WHILE_ARRAY	24.87	24.83	24.64	24.77	+8.63%
FOR_GEN_LIST	29.83	29.39	29.66	29.62	+7.37%
WHILE_DO_GEN_LIST	28.63	29.65	29.12	29.13	+9.25%
DO_WHILE_GEN_LIST	30.02	29.61	29.39	29.67	+3.48%

Table 4. Results after caching and loop unrolling, with clean-up code

Method description	First run	Second run	Third run	Average	Percentage
FOR_ARRAY	24.59	24.73	24.93	24.75	+9.00%
WHILE_DO_ARRAY	24.87	24.71	24.46	24.68	+8.49%
DO_WHILE_ARRAY	24.90	24.78	24.63	25.77	+4.94%
FOR_GEN_LIST	29.42	29.20	29.29	29.30	+8.38%
WHILE_DO_GEN_LIST	29.93	28.98	29.14	29.35	+8.56%
DO_WHILE_GEN_LIST	29.12	29.27	29.31	29.23	+4.91%

These factors collectively lead to lower overhead and better predictability in array-based iterations.

5.2. PRACTICAL IMPLICATIONS

From a practical perspective, the results indicate that manual optimisation techniques can yield tangible benefits, but their effectiveness is context-dependent. While caching reliably improves performance with few downsides, loop unrolling provides more limited and situation-specific advantages. Notably, the relatively modest improvements (up to ~10%) suggest that such optimisations should be applied judiciously, primarily in performance-critical sections of code. In many instances, the importance of readability and maintainability may surpass the advantages of micro-optimisation. Overall, these findings underscore the necessity of empirical evaluation. Even well-established optimisation techniques can produce varying results based on data structures, runtime behaviour, and compiler optimisations, making benchmarking an essential practice in performance engineering.

The findings of this study should be interpreted with caution when applied to real-world systems. While micro-optimisations can provide measurable improvements, their overall impact depends on the broader application context. In many cases, higher-level design decisions-such as algorithm selection, data structure choice, or system architecture have a far greater influence on performance than low-level loop optimisations. Therefore, developers should prioritise optimisation efforts based on profiling and empirical evidence, rather than relying solely on theoretical expectations or isolated microbenchmarks.

6. ADVANCED ANALYSIS

6.1. INFLUENCE OF CPU ARCHITECTURE AND MEMORY HIERARCHY

This study goes beyond surface observations to examine performance through the lens of CPU architecture and memory hierarchy. Modern processors function like orchestra conductors, effectively managing multi-level cache systems—L1, L2, and L3—to minimise memory access latency. Arrays benefit from contiguous memory allocation, which enhances spatial locality and optimises cache line efficiency. As a result, when traversing arrays sequentially, the CPU prefetches data efficiently, reducing cache misses and increasing throughput. In contrast, generic lists, while array-backed, add layers of abstraction that can hinder optimal memory access. Even small overheads, such as extra bounds checks or method calls, can disrupt instruction pipelining and diminish cache efficiency.

6.2. BRANCH PREDICTION AND CONTROL FLOW OPTIMISATION

Another important factor in loop performance is branch prediction. Modern CPUs use sophisticated branch predictors to lessen the impact of conditional jumps. Loops with clear, predictable control flow, such as for and while loops, are well-served by accurate branch predictions. Loop unrolling can further reduce conditional branches, enhancing performance by sidestepping the issues associated with branch misprediction. However, the findings suggest that this advantage may be limited in practice, likely because current processors and JIT compilers effectively optimise common branch patterns.

Interaction with JIT Compiler Optimisations The .NET JIT compiler plays a significant role in the final performance of the code we evaluated. Techniques like loop hoisting, bounds-check elimination, and method inlining are automatically applied at runtime. The modest benefits from manual optimisations indicate that the JIT compiler already handles a large portion of these enhancements. Thus, manual strategies such as loop unrolling may yield diminishing returns, especially for simpler loop structures. This interaction highlights a crucial point: optimisation methods that work well in low-level languages may not always be effective in managed environments, where the runtime actively optimises and refines code.

6.3. STATISTICAL STABILITY AND MEASUREMENT RELIABILITY

While the results are based on averaged execution times, we must consider the stability of our measurements. BenchmarkDotNet performs multiple warm-ups and measurement iterations, using statistical techniques to filter out noise and eliminate outliers. The small variance observed suggests our measurements are stable and reliable. However, even slight environmental changes—like background processes or CPU frequency fluctuations—can affect results. This underscores the need for controlled benchmarking environments and careful interpretation of microbenchmark outcomes.

6.4. LIMITATIONS OF THE STUDY

While this study provides valuable insights, it has several limitations to consider. Firstly, our dataset consists of simple objects with minimal internal complexity. Although this allows for controlled experimentation, it does not capture the complexity of real-world object graphs, memory fragmentation, or varying access patterns. Secondly, our experiments were conducted on a single hardware and software configuration. Different processors, cache architectures, or runtime versions may behave differently, which limits the generalisability of our findings. Thirdly, we explored only a limited range of optimisation techniques. Other methods, such as parallelisation, vectorisation, or alternative data structures, could have a more substantial impact on performance.

6.5. GENERALISATION AND PRACTICAL RELEVANCE

The insights from this study should be applied to real-world systems cautiously. While micro-optimisations can lead to noticeable improvements, their overall impact depends on the wider application context. Often, higher-level design choices—like algorithm selection, data structure decisions, or system architecture—have a more significant effect on performance than low-level loop optimisations. Therefore, developers should base their optimisation efforts on profiling and empirical evidence rather than solely on theoretical expectations or isolated microbenchmarks.

7. CONCLUSION

This paper presents an experimental evaluation of loop performance in C#. It specifically focuses on collections of reference-type objects. The results indicate that loop efficiency is influenced by the choice of iteration constructs. Furthermore, optimisation techniques and the underlying data structure play a significant role in performance.

The findings suggest that although the baseline performance differences between loop constructs are minor, employing optimisation techniques like invariant expression caching and loop unrolling can result in significant improvements, with some cases up to 10%. In particular, for loops consistently perform well when traversing arrays and while loops can compete effectively with generic lists under suitable conditions.

These results suggest that performance characteristics are highly context-dependent and should not be generalised without empirical validation. Key factors influencing execution efficiency include memory access patterns, JIT compiler behaviour, and the implementation of data structures. Practically, this study demonstrates that optimisation techniques driven by developers must be paired with an understanding of system behaviour during runtime. Decisions affecting performance should be grounded in thorough benchmarking and analysis rather than assumptions. Future research could expand this work by utilising larger and more diverse datasets, evaluating additional data structures, and exploring the interaction of JIT optimisations with modern hardware architectures. Furthermore, examining the impact of high-level abstractions like LINQ on performance would provide valuable insights.

REFERENCES

[1] J. Albahari and B. Albahari, "C# 10 in a Nutshell", 10th ed., Sebastopol, CA, USA: O'Reilly Media, 2022.

[2] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, and Tools", 2nd ed., Boston, MA, USA: Pearson, 2006.

[3] S. Muchnick, "Advanced Compiler Design and Implementation", San Francisco, CA, USA: Morgan Kaufmann, 1997.

[4] H. S. Warren, "Hacker's Delight", 2nd ed., Boston, MA, USA: Addison-Wesley, 2013.

[5] J. Skeet, "C# in Depth", 4th ed., Shelter Island, NY, USA: Manning Publications, 2019.

[6] A. Troelsen and P. Japikse, "Pro C# 10 with .NET 6", 11th ed., New York, NY, USA: Apress, 2022.

[7] A. Akinshin, A. Sitnik, Y. Stepanov and T. Cassell, "Overview.", BenchmarkDotNet, [Online]. Available: https://benchmarkdotnet.org/articles/overview.html. [Accessed 01 03 2026].

[8] J. Duffy, "Concurrent Programming on Windows", Boston, MA, USA: Addison-Wesley, 2008.

[9] J. Richter, "CLR via C#", Redmond, WA, USA: Microsoft Press, 2010.

[10] M. L. Scott, "Programming Language Pragmatics", 4th ed., Burlington, MA, USA: Morgan Kaufmann Publishers, 2015.

[11] A. Akinshin, "Pro .NET Benchmarking", New York, NY, USA: Appress, 2019.

[12] J. Alls, "Clean Code with C#", 2nd ed., Birmingham, UK: Packt Publishing, 2023.