Tampere University

Tapio Löppönen

BYTECODE INSPECTION FOR PERFORMANCE IMPROVEMENT

M. Sc. thesis
Faculty of Information Technology and Communication Sciences
April 2022

ABSTRACT

Tapio Löppönen: Bytecode Inspection for Performance Improvement
M. Sc. thesis
Tampere University
Master's Degree Programme in Software Development
April 2022

The .NET virtual machine provides support for running cross-platform applications. The virtual machine provides cross-platform support by translating platform-independent bytecode into platform-dependent instruction during execution. The bytecode, in our case, the Common Intermediate Language, is generated by compiling C# code into the bytecode. By compiling a high-level language into bytecode, the application can be developed using a high-level language, compiled once, and deployed on multiple platforms. The .NET virtual machine uses stack architecture, which requires the C# to be converted into a stack-based bytecode during compilation. In addition to producing stack-based bytecode, the compiler removes syntactic sugar from the code. Syntactic sugar is removed by replacing complex semantics with simpler ones.

In this thesis, we performed benchmarks to determine the performance effects of high-level changes. In addition to benchmarks, we inspected the changes on the bytecode level. The benchmark methods iterate over a sequence of elements, counting the sum of all elements. We started by comparing the performance differences between arrays and lists. The list showed a small overhead in most cases and a noticeable overhead in some cases. Because the generic list contains different implementation details, we performed additional benchmarks on the used syntactic sugar. The additional benchmarks included comparing the performance of fields, properties, and indexers. In most cases, simple syntactic sugar does not add overhead to the performance. However, using properties and indexers through a field inside an instanced method seems to add overhead. The overhead could be removed by storing the target object in a local variable before the loop. Storing the target object in a local variable removes the need to load it from a field during each iteration.

Keywords: .NET, C#, CIL, Compiler Theory, Micro-optimization

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

.NET	(dot net) is a stack-based process virtual machine
C#	(c sharp) is a high-level programming language
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CTS	Common Type System
GC	Garbage Collection
JIT	Just-In-Time
JVM	Java Virtual Machine

1 INTRODUCTION

Software development is an ever-growing industry that keeps expanding as new devices and platforms are introduced. This poses a challenge to software development since different platforms support different things. Traditional programming languages that are compiled into native code can be executed on certain platforms. This makes software development harder since platform-specific changes may be required for an application to work. Process virtual machines solve the problem by providing an execution environment to execute bytecode. The execution environment translates bytecode instructions into native code. Using a bytecode language as an intermediate format provides a middle ground between high-level code and native code. Commonly, bytecode is generated by compiling a high-level programming language into a bytecode language. In addition to code execution, the process virtual machine can utilize a garbage collector to manage memory.

Applications executed inside a process virtual machine are used for all kinds of applications. These include console, web, Internet of Things, mobile, and desktop applications. The main limitation is system programming, where the overhead caused by the virtual machine and garbage collection becomes problematic. However, in regular applications, this overhead is not noticeable. The time lost in slower execution is saved in the shorter development time.

Bytecode-based applications go through two main stages. The first stage contains compiling a high-level programming language into bytecode. In this stage, a compiler translates a high-level programming language into a bytecode language. This translation includes modifying the code structure to fit the virtual machine architecture. The second stage includes executing the bytecode inside a process virtual machine. During the execution, the bytecode is translated into native instructions. The translation can be done with an interpreter or a just-in-time compiler.

This thesis is a look at Common Language Runtime, which is an implementation of the Common Language Infrastructure standard. The standard defines a stack-based virtual machine and executable code. We will also look at the C# (c sharp) programming language and how it is compiled into the executable code. Finally, we will perform benchmarks to determine the performance impact of high-level changes. We will compare the performance between an array and a generic list. Additionally, we will compare the performance of fields, properties, and indexers to access values. The performance of each feature is tested with multiple methods using a benchmarking framework. In addition to the high-level changes, we will inspect the bytecode. Finally, the benchmark results and bytecode are used to inspect the impact of high-level changes. Code examples and benchmarks used during this thesis are available on GitHub (Löppönen, 2022).

This thesis aims to inspect the changes made during bytecode compilation and how they affect the overall performance of the resulting application. The compilation process includes transforming C# code into stack-based bytecode. This transformation also includes removing the syntactic sugar included in C#. Syntactic sugar is removed by replacing the higher-level syntax with semantically matching lower-level syntax. The replacement process can produce varying amounts of bytecode. We will also inspect the changes produced by high-level syntactic refactoring.

- RQ1. What is the performance impact of syntactic sugar?
- RQ2. What is the performance impact of C# loop statements?
- RQ3. What type of high-level modifications can be used to improve performance?

The second chapter introduces a few background knowledge areas used in the following chapters. It includes an introduction to process virtual machines, compilers, and disassembling. The third chapter introduces the Common Language Runtime, which defines an environment for executing applications. The chapter includes information about the type system, metadata, and generics. The fourth chapter introduces the .NET runtime, which is an implementation of the Common Language Runtime. It includes a better look at various implementations of the runtime. These implementations include the JIT compiler, Garbage Collector, and standard libraries. The fifth chapter contains a few examples of compile-time transformations taking place during C# to bytecode compilation. These examples include lowering, for-each statement, and properties. The sixth chapter contains the research task and results. The final chapter contains the conclusion of the thesis.

2 BACKGROUND

The journey from high-level code to code execution inside a process virtual machine is a multi-step process. This multi-step process contains multiple knowledge areas. In addition to code execution, additional steps may be required to inspect the changes on a bytecode level during the process. The process starts with compiling high-level code into bytecode. During compilation, the high-level code is translated into bytecode. The compilation can contain various optimization techniques used to optimize the resulting bytecode. In the end, the resulting bytecode is packaged into a binary file. The binary file can be disassembled for human inspection or given to a process virtual machine for execution. Process virtual machines can be used to execute bytecode. They provide an execution environment to execute code and a garbage collection to manage memory during the execution. Both the execution environment and garbage collector can contain implementation details that affect the code execution.

2.1 Process Virtual Machines

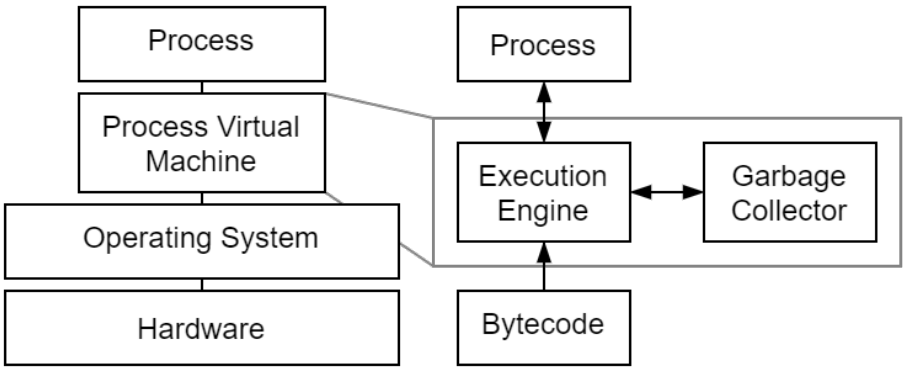

Figure 1. *Process Virtual Machine.*

A process virtual machine or application virtual machine, shown in Figure 1, is designed to run a single program with a single process. It runs just like a regular application within the host OS as a process. The virtual machine is created when a process is initiated and destroyed when the process exits or dies (Sudha et al., 2013). A process virtual machine is made out of multiple components. These components can be divided into two units based on their responsibilities. These units are the execution engine and garbage collector. The execution engine is responsible for loading and executing code (ECMA International, 2012, p. 72; Kokosa, 2018, p. 239). Code can be executed using an interpreter, just-in-time compiler, or a combination of both (Shi et al., 2008). Additionally, the execution engine can provide additional features such as a unified type system, exception handling, and additional security features (Costa & Rohou, 2005; Kokosa, 2018,

p. 238). The garbage collector is responsible for allocating and releasing memory (ECMA International, 2012, p. 6; Kokosa, 2018, p. 239).

The execution environment translated platform-independent bytecode into platform-specific machine code. Meijer & Gough (2000) note that by using an intermediate language, you need only $n + m$ translators instead of $n * m$ translators, to implement n languages on m platforms. The bytecode is produced by compiling high-level code into bytecode. A bytecode language is easier to translate into platform-specific instructions compared to high-level programming languages. Bytecode languages can also provide language independence by defining a shared type system (Costa & Rohou, 2005; Meijer & Gough, 2000). A shared type system allows different tools to share the declaration, using, and managing of primitive and user-defined types.

Popular process virtual machines such as Java Virtual Machine and .NET use a virtual stack architecture. Stack architecture provides a small code size and does not require explicit registers to be defined (Maierhofer & Ertl, 1997; Shi et al., 2008). Stack architecture executes code by pushing values onto an evaluation stack. A single instruction will either push a value onto the stack or pop values off the stack. Values on the stack are consumed by their first use and are thus unavailable for subsequent uses (Park et al., 2011). Values with multiple uses require multiple copies. In addition to consuming values on first use, the values must be pushed onto the stack in the expected consumption order as random operand access is not supported (Park et al., 2011). An alternative to stack architecture is virtual register architecture. Virtual register architecture uses virtual registers to store values. In this approach, the bytecode defines registers where the values are stored. Shi and others (2008) suggest that stack-based bytecode is better suited for JIT compilation since it does not make any assumptions about the number of available registers. They also found that register-based bytecode outperforms stack-based bytecode when interpreted.

2.2 Compilers

Compilers are used to translate one language to another. Commonly a high-level language is compiled into a lower-level language. When targeting a process virtual machine, a high-level language is compiled into a bytecode language used by the virtual machine. There may be multiple high-level programming languages that target a single bytecode language. Java, Kotlin, and Scala are high-level programming languages that are compiled into Java Bytecode, which is used by Java Virtual Machine (Urma, 2014). C#, F#, and Visual Basic are high-level programming languages that are compiled into Common Intermediate Language (CIL), which is used by .NET (.NET Programming Languages, 2022).

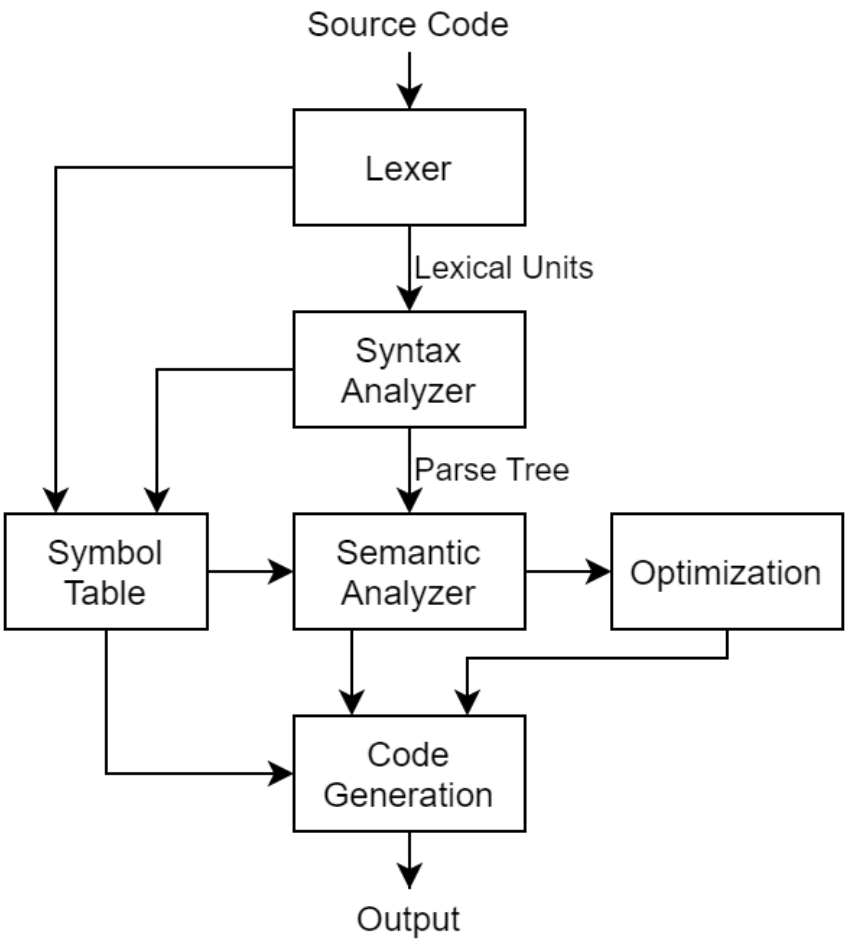

Figure 2. *Compilation process using a compiler.*

Compilers compile the source code in multiple steps. The steps are visualized in Figure 2. The compilation process starts with lexical analysis, which is used to split the source code into lexical units. These lexical units are formed based on the syntactic rules of the language. Each lexical unit represents a single identifier, special word, operator, or punctuation symbol. Once the source code is converted into a list of lexical units, the list is given to a syntax analyzer. The syntax analyzer is used to build a parse tree from the lexical units. The parse tree is a hierarchical tree structure representing the syntactic structure of the source code. During the parse tree generation, a symbol table is generated. The symbol table contains the various identifiers present in the parse tree. Each identifier stored in the symbol table references the defining parse tree node. This allows the following steps to access various parts of the parse tree through the identifiers stored in the symbol table. After the parse tree and symbol table are created, they are given to a semantic analyzer which performs various checks on the parse tree. Semantic analysis includes checking that the used identifiers have been defined and that they follow the correct typing rules. After semantic analysis, additional optimization can be applied. Finally, the parse tree is used to generate code. (Sebesta, 2012, pp. 24–27)

2.3 Disassembly

Bytecode-based applications are commonly compiled directly into a binary format used by the process virtual machine. Storing the application as a binary file provides several benefits. Binary files provide fast access to the data. Values, character, and miscellaneous data can be stored and accessed without additional conversion being required. Sometimes the stored data may be hard to represent in a text format, which makes storing bytes a more accessible solution. Binary files also provide smaller file sizes. The main disadvantage of binary files is that they are not human-readable. A disassembler can be used to convert a binary file into a text format. Converting a binary file into a text format allows the bytecode to be inspected. Additionally, an assembler can be used to compile the text format back into binary format. This provides a platform to modify compiled applications without having access to the source code. This can be useful when working with applications where access to the source code is lost or unknown.

When bytecode applications are executed using a JIT compiler, the JIT compiler produces machine instructions. The machine instructions produced by the JIT compiler can be disassembled for further inspection. Inspecting the machine instructions produced by the JIT compiler can provide further insight into the application's logic. However, when inspecting code generated by a JIT compiler, it is important to understand how the code was generated. There are different types of JIT compilers, which can produce different results. For example, some JIT compilers recompile often used methods, which can change the outcome. Additionally, it is important to note that the JIT produces platform-specific results, and the changes made based on the compiled instructions may be hard to transfer to different platforms.

3 COMMON LANGUAGE RUNTIME

Common Language Runtime (CLR) is a Virtual Execution System published by Microsoft. It is an implementation of the Common Language Infrastructure (CLI), which was also initially designed by Microsoft. The CLI is a public standard, ECMA-335 Common Language Infrastructure, that defines a virtual execution system and executable code (ECMA International, 2012). Having a public standard as the basis for a runtime allows different vendors to provide their own implementations of it while still allowing the executable code to be shared between the different implementations. Using a virtual execution system allows the same code to be deployed on multiple platforms, only requiring the execution system to be implemented on the target platform. Popular implementations of the CLI standard include .NET and Mono.

The Common Language Infrastructure took heavy inspiration from Java Virtual Machine (JVM) and its ecosystem. It is a few years younger than JVM, which allowed the CLI design team to use JVM as a comparison when designing the different components. JVM was originally designed to be a virtual machine for Java bytecode applications. This made it a good platform to run object-oriented Java applications but made it harder to support other programming languages and programming paradigms. The CLI was designed from the start to support a wide range of programming languages (Meijer & Gough, 2000). CLI aims to provide support for object-oriented, functional, and procedural programming paradigms. This allows multiple high-level languages to target the CLI.

When building a CLI application, it is converted into a set of binaries, including the project and its dependencies. These binaries include the intermediate language presentation of the application, with additional files that are required by the application. These binaries can form an executable file or a library file. Using a binary format to represent the build application allows the same binary to be used in different environments. Using a standardized binary format to store the intermediate language allows the application to be decompiled back into the intermediate language if required. This allows the compiler-generated code to be inspected. In addition to this, the application can be decompiled, modified, and recompiled without access to the original source code. This can be useful when working with legacy systems or applications where the original source is lost or unknown.

3.1 Common Type System

At the center of the CLI is a unified type system, the Common Type System (CTS), that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-language integration, type safety, and high-performance code execution

(ECMA International, 2012, p. 6). By sharing a common type system and high-level execution environment, interoperability between different languages becomes easier than binary interoperability. Easy interoperability is a prerequisite for multi-language library design and software component reuse (Meijer & Gough, 2000).

Type safety is promoted by typing everything. It guarantees that references are typed, and the value or object referenced is typed. It also guarantees that only appropriate operations can be invoked. This includes making sure that the method or field for the given type exists, with valid visibility based on where the reference is compared to the referenced entity.

The common type system is designed to support object-oriented, function, and procedural programming languages. The type system revolves around two entities: objects and values. Value types are used to represent simple bit patterns for things like integers and floats. Each value also stores a type that defines what the bits represent and what operations can be performed on them. Objects are self-typing entities with unique identities that contain slots to store other entities. While the content of the slots may change, the identity of the object never changes. (ECMA International, 2012, p. 16)

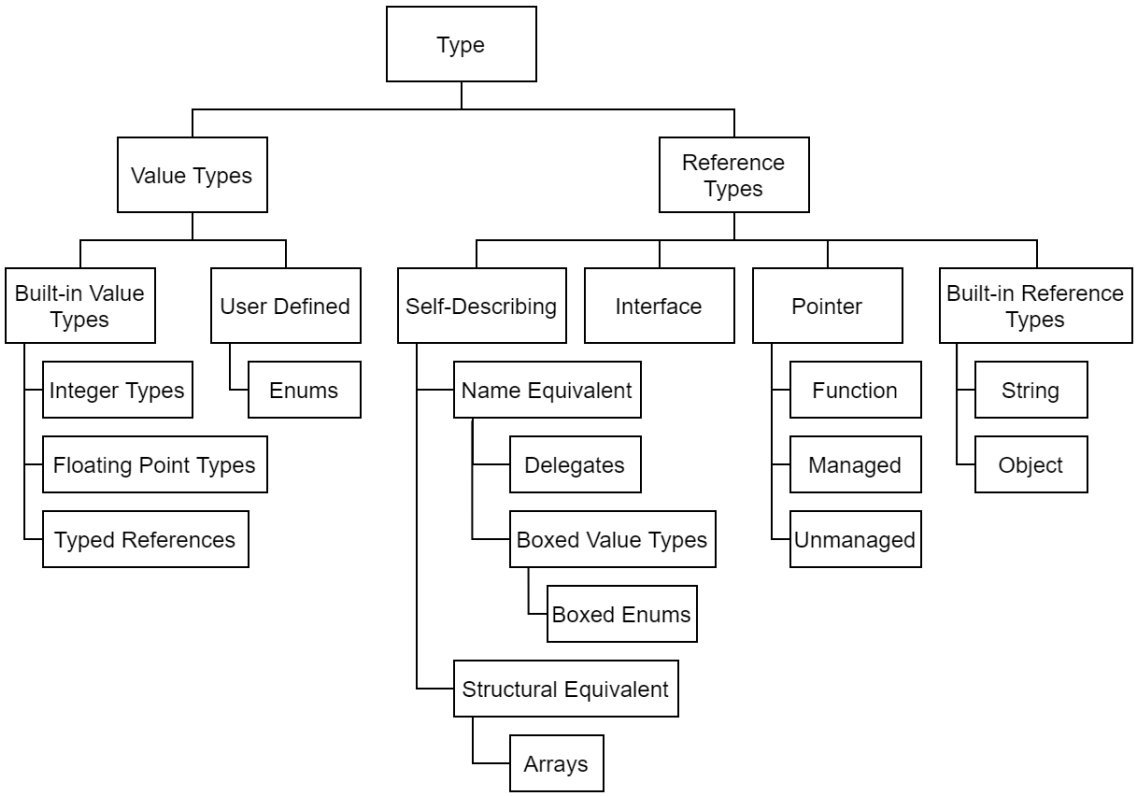

Figure 3. *Common Type System.*

Figure 3 displays the type system (ECMA International, 2012, p. 18). The type system can be used to determine how a value is stored within the virtual execution system. A value of a reference type is always allocated on the heap (Fruja, 2008). This results in variables with reference type pointing to values on the heap. Additionally, the values are treated as pass-by-reference. Values of a value type are allocated either on the evaluation

stack or on the heap within an object class instance (Fruja, 2008). Variables with value type store the value directly and treat the values as pass-by-value.

The common type system is designed to be future-proof, with easy to expand types. This means that it should be easy to add new types without affecting the existing types or their implementation. For example, signed integers are stored as `int16`, `int32`, and `int64`. If a 128-bit integer is required in the future `int128` could be added without breaking anything in the existing implementation. In contrast, C# contains keywords for the values as `short`, `int`, and `long`, which do not explicitly state their implementation sizes.

The standard does not define memory layouts for object fields. However, the bytecode contains keywords for defining different layouts. The first option is automatic, which allows the runtime to determine the best layout. This allows the runtime to reorder the fields as seen fit. Retaining rights to reorder fields allows the runtime to store objects in a compact way with aligned fields. Additional keywords include manual field offsets and sequential memory layouts. Different memory layouts may be required when communicating with unmanaged code.

Field alignment is part of data structure alignment that is concerted in the way that the data is arranged and accessed in the memory. A Processor reads the memory in word-sized chunks that are defined based on the processor's architecture. A word is the natural unit of data for the processor. Common word sizes are 4 bytes for 32-bit architecture and 8 bytes for 64-bit architecture (Kokosa, 2018, p. 9). Additionally, the data is addressed at memory locations that are multiples of the word size. Providing fast access to values stored in these chunks requires the values to be aligned with the chunks (Eimouri et al., 2016). This means that each value should start at the start of the chunk, or multiple smaller values should be distributed evenly in a single chunk. Uneven values are slower to access since they require additional reads to access. The data can be aligned by applying padding after value so that the following value can start at the beginning of the following chunk (Eimouri et al., 2016). Hence, allowing the runtime to reorder fields should provide the best possible memory layout in different environments.

3.2 Metadata

Metadata is used to store additional information about the bytecode. It is used to describe and reference types defined by the common type system. It allows data to be stored in a neutral format that is not directly tied to any specific programming language. This makes the bytecode easier target for multiple high-level programming languages and different tools since they can use the language-independent metadata to communicate with each other. The metadata is produced by a compiler when a high-level language is compiled into bytecode. Costa & Rohou (2005) note that the metadata is not optional information added for the convenience of other tools. It is necessary for the proper execution of the code. They also found that metadata is a substantial part of the total bytecode size.

New types – value types and reference types – are introduced into the CTS via type declarations expressed in metadata. In addition, metadata is a structured way to represent all information that the CLI uses to locate and load classes, lay out instances in memory, resolve method invocations, translate CIL to native code, enforce security, and set up runtime context boundaries. (ECMA International, 2012, p. 53)

Metadata is also used to form CLI components and assemblies. CLI components are used to divide the application logic into smaller software components. Each component contains metadata about the declarations, implementations, and references made or required by the component. This makes each component self-describing since they do not require external sources to describe them. When deploying an application, CLI components and additional files are packed into assemblies, which are logical units of functionality. Assembly can be an executable file or linked library. The main difference is that the linked library does not contain an entry point, which prevents it from being executed by itself. A single project can contain multiple assemblies. The compile-time of large projects can be reduced by dividing them into multiple assemblies. Multiple assemblies divide the compilation into smaller chunks. This also means that only the modified assembly needs to be recompiled.

3.3 Generics

The CLI standard also defines generics. Generics can be used to define types and methods that do not explicitly state their type. Rather than having an explicit type, they contain a type parameter, which defines the type when the type or method is created or invoked. This increases the code reusability since the generic types can be used to implement reusable data structures, algorithms, and other components. Reusability reduces code duplication and errors caused from maintaining multiple typed data collections (Parnin et al., 2012). The addition of type parameters also allows type safety to be enforced since the type parameter can be used for type checking. Fruja (2006) concludes that generic types maintain the type safety of the CLR.

Type parameters may be constrained so that they can only be instantiated with types having particular characteristics. Type constraints include whether a type is a reference type or a value type, whether it derives from a specific class or implements a particular interface (or interfaces), and whether it supplies an accessible no-arg constructor (Brosgol, 2010). In practice, this means that a variable with the generic type can use fields and methods described by the type constraint. Generics provide a small runtime overhead compared to casting and boxing because the type conversions and checks can be done during compile time. Brosgol (2010) concludes that runtime overhead is avoided after the initial JIT compilation.

4 .NET

.NET is Microsoft's implementation of the CLR. It has evolved during the years with the CLI and CLR. The original implementation was the .NET Framework, which is a Windows-only implementation of the runtime (Akinshin, 2019, p. 96). In 2016 a new runtime was introduced named .NET Core which is a cross-platform implementation of the runtime (Akinshin, 2019, p. 100). .NET Core provides partial support for .NET Framework applications on Windows, OSX, and Linux, but there is no guarantee that .NET Framework applications could run in .NET Core. In 2020 a new .NET Core version was released with only the name .NET (What's new in .NET 5, 2022). This release was the first step to combine .NET Framework and .NET Core into a single runtime.

Mono is an open-source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime (Mono, 2022). It tries to provide cross-platform support for the otherwise Windows-only .NET Framework applications. However, some functionality is missing compared to .NET (Mono Application Portability, 2020). Mono is sponsored by Microsoft (Mono, 2022) and is used by various companies, including the Unity game engine (Companies using Mono, 2020). Since Mono is a separate implementation of the standard, it may yield different performance results compared to .NET.

Optimizing .NET applications may yield different results on different versions. In addition to the .NET release version, it is important to note that the different components making up the virtual machine contain versions. Primarily these include the JIT compiler, garbage collector, and standard libraries.

4.1 RyuJIT

The .NET virtual machine uses just-in-time compilation to execute code. .NET 5 uses a just-in-time compiler named RyuJIT. It was initially introduced in 2013 (.NET Team, 2013) and took over the previous implementation in 2018 (Forstall, 2018). RyuJIT aimed to improve optimization, code generation, and to open up the design and implementation.

The JIT compilation consists of different phases. The initial phase consists of importing the CIL bytecode and transforming it into an intermediate representation. The intermediate representation is then prepared for the optimization phases. During the optimization phases, a range of optimization techniques is applied to the intermediate representation. Finally, the back-end phases are used to emit native code based on the intermediate representation. (RyuJIT Tutorial, 2021)

RyuJIT executes code by compiling method bodies into native code and then running the native code. The code is JIT compiled as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context

of that process (Managed Execution Process, 2021). Unused methods are not compiled. RyuJIT uses a single-tier approach for code execution (RyuJIT Tutorial, 2021). This means that all used methods are JIT-compiled, and the JIT compilation is done only once. Other approaches include interpreting rarely called methods to remove overhead caused by the initial JIT phases and multi-tier JIT compilation. Multi-tier JIT tries to improve performance by recompiling often used methods. Multi-tier JIT for RyuJIT has been in the works but is not part of any stable release.

4.2 Garbage Collection

The .NET virtual machine uses a garbage collector to allocate and release memory on a heap. The heap is divided into a small object heap and a large object heap. By default, the small object heap stores object smaller than 85000 bytes, and the large object heap stores all the larger objects. However, the large object heap size threshold can be modified. A new heap allocation is made when a new object of reference type is created. The common type system can be used to determine if a given object is a reference type.

The garbage collector is a generational garbage collector, which means that the objects stored in the memory are divided into generations. Using generations to divide the objects allows the garbage collector to process the objects in smaller chunks. During collection, the garbage collector tries to find and release objects no longer referenced by the active program. Objects that are referenced by the active program are considered living, and the objects no longer referenced dead.

In total, there are three generations. The generations use zero-based indexing. Generation 0 stores all the newly created small objects. Most of the objects are expected to die during the first generation. Generation 1 and 2 are used to store long-living objects from the previous generations. However, generation 2 is the last generation meaning that the object can not move to another generation. An object is long-living if it is alive during collection. A generation gets collected when it requires more space. The first generation is mostly for short-living objects since it gets collected when enough new objects have been created. During collection, the living objects are moved to the next generation. The next generation will be collected when it requires space. Large object heap has a single generation, which is collected alongside generation 2. Collecting generation 2 is considered a full garbage collection since every generation is collected.

4.3 Class Libraries

Class libraries are used to share common functionality between multiple applications. They provide high interoperability between different programming languages using the common language specification (CLS). This allows the class libraries to be used by any CLS-compliant compiler. The common language specification is a subset of the common type system.

Class libraries are divided into three types. The first type is platform-specific class libraries, which provide support to platform-specific APIs. The second type is portable class libraries, which provide platform-independent APIs. Finally, the third type is .NET Standard, which is a combination of portable and platform-specific APIs. The .NET Standard is a formal specification of APIs available on multiple implementations. However, .NET 5 adopts a different approach to establishing uniformity, and this new approach eliminates the need for .NET Standard in many scenarios (.NET Standard, 2021). However, it should be noted that .NET Standard is not deprecated. .NET Standard is still needed for libraries that can be used by multiple .NET implementations (.NET Standard, 2021). Multiple implementations including, .NET Framework, .NET Core, and Mono, provide support for the .NET standard.

5 BYTECODE COMPILATION

Compiling a high-level programming language into bytecode is an important part of the overall development process. It is important to understand what kind of bytecode is generated from a high-level programming language since the bytecode gets executed during runtime. Optimizing high-level code without understanding what kind of bytecode it generates can lead to redundant changes. Additionally, some high-level language features are converted into a different format. These language features are known as syntactic sugar, which means that they provide simplified syntax for implementing existing functionality (Syntactic Sugar - Wikipedia, 2021). The syntactic sugar is removed during compilation by using lowering (Warren, 2017). This means that the resulting bytecode may be different from the original high-level programming language. Understanding the differences between the high-level programming language and bytecode should help with understanding how the code is executed. Additionally, bytecode inspection can be used to gain better insight into the application logic.

C# is a high-level object-oriented programming language designed as the flagship language for the Common Language Infrastructure. It is based on a public standard, ECMA-334 C# Language specification (ECMA International, 2017). The standard contains definitions for the language syntax, representation, and semantic rules of C# programs. The C# Language specification does not contain anything related to the Common Language Infrastructure. Roslyn is a popular .NET Compiler Platform containing different APIs to compile, analyze, and refactor C# and Visual Basic code. The Roslyn compiler compiles C# and Visual Basic into CIL bytecode.

C# and CIL are both object-oriented programming languages. However, CIL uses explicit syntax to store information, whereas C# contains implicit declarations. For example, C# contains default access modifiers, CIL does not. C# classes inherit an object class without explicit inheritance, CIL classes extend it explicitly. Every class in both languages is based on the object class. This means that they must inherit the object class at some point. Additionally, the CIL bytecode uses different syntax rules for identifiers. This allows the C# compiler to generate bytecode identifier, which would be illegal in C#. This makes overlap between the user and compiler-generated identifiers impossible.

Compiler back-ends for stack machines perform a DFS (depth-first search) traversal on the parse tree and generate code as a side effect. Common subexpression elimination finds expressions with multiple uses. The resulting DAGs (directed acyclic graph) complicate stack code generation since operands on the stack are consumed by their first use and are thus unavailable for subsequent uses. DAGs can be converted to trees by storing multiply-used values in temporaries and replacing references by references to the corresponding temporaries. (Park et al., 2011)

5.1 Hello World

Creating a simple Hello World application in C# requires only a few lines of code. An example of this can be seen in Code example 1. It contains using-directive, class declaration, main method, and method call. The using-directive is used to import the Console class, which prints "Hello World" into a terminal. The compiler makes the static Main method into the application entry point by default.

```
using System;
public class HelloWorld {
    public static void Main() {
        Console.WriteLine("Hello World");
    }
}
```

Code example 1. *Simple Hello World application in C#.*

A compiled version of the Hello World application can be seen in Code example 2. The bytecode version follows a similar class structure to C#. However, the class and method declarations contain additional information, and the bytecode class contains an explicit constructor and extends a `System.Object` class. The methods bodies start with various keywords. However, the main bytecode instructions are prefixed with `IL_` numbering. These instructions are translated into native instructions by the JIT compiler.

All CIL bytecode method bodies start with a `.maxstack` keyword. The keyword describes how many items the method may push onto the evaluation stack. The max stack size must be defined ahead of time. Commonly the max stack size is calculated by a compiler during bytecode compilation. Items pushed onto the evaluation stack are placed in stack slots. A single slot does not have a fixed size. This means that the value can not be used to determine the evaluation stack frame size. However, the value is used to determine how many values need to be tracked by an analyzer during the methods execution.

In the Code example 2, the Main method contains an additional `.entrypoint` keyword, which marks the entry point for the application. The keyword is used because the bytecode does not make any assumptions about the language or its entry point. The C# compiler uses a static Main method as an entry point, which is why it placed the keyword into the bytecode. Providing the C# compiler with a different entry point would move the keyword in the bytecode.

Executing the bytecode would start by pushing the Main method onto the stack. The Main method would be selected as the starting point based on the `.entrypoint` keyword. The method would be pushed onto the stack as a stack frame. The stack frame contains zero arguments and local variables. Additionally, it contains an empty evaluation stack with eight stack slots, shown by the `.maxstack` keyword. The first instruction inside the Main method would push the string "Hello World" onto the evaluation stack. The next instruction would call the static WriteLine method, which would take the previously pushed string as an argument. The called method would be added as a stack frame onto the stack, and it would consume the string from the Main methods evaluation stack. Once the called method is executed, the stack frame would be removed from the stack,

and the execution would return to the Main method. The last instruction would return the method. Since the Main method is the entry point, the application would exit because there are no more instructions to execute.

```
.class public auto ansi beforefieldinit HelloWorld
    extends [System.Runtime]System.Object
{
    .method public hidebysig static
        void Main () cil managed
    {
        .maxstack 8
        .entrypoint
        IL_0000: ldstr "Hello World"
        IL_0005: call void
           [System.Console]System.Console::WriteLine(string)
        IL_000a: ret
    }

    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed
    {
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: call instance void
           [System.Runtime]System.Object::.ctor()
        IL_0006: ret
    }
}
```

Code example 2. *Compiled Hello World application.*

5.2 Lowering

Lowering (Warren, 2017) or desugaring (Syntactic Sugar - Wikipedia, 2021) consists of internally rewriting more complex semantic constructs in terms of simpler ones. Lowering is used during compilation to convert high-level code into lower-level code. Converting the high-level code to a low-level code allows the compiler to reuse the logic used to implement the lower-level logic. Lowering can also help with avoiding compiler bugs and edge-cases (Warren, 2017). The Roslyn compiler contains a sub-directory dedicated to lowering (Lowering - dotnet/roslyn, 2022).

C# implements various keywords to implement loops, including `for`, `foreach`, `while`, `do`, and `goto` statements. Inspecting the bytecode generated from `for`, `while`, and `goto` loops displays that they generate matching bytecode. Additionally, a `foreach` loop can be lowered into a `while` loop. This means that all the different loops can be expressed using a `goto` statement. The CIL bytecode uses branch instructions to jump to a given location. This matches the behavior of `goto` statements. Figure 4 displays the lowering of various loops statements in the Roslyn compiler (Lowering - dotnet/roslyn, 2022).

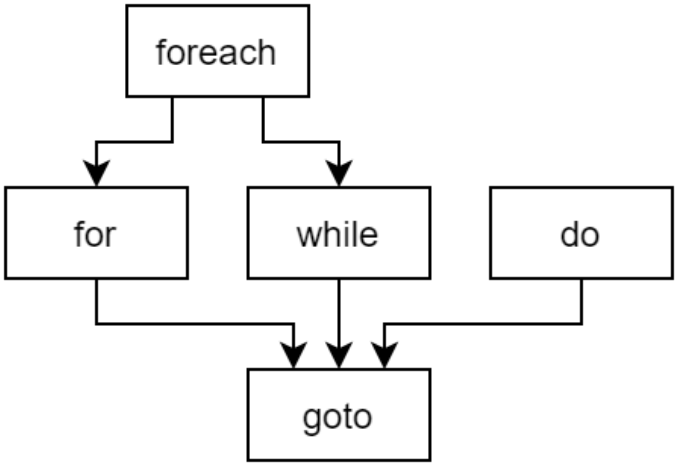

Figure 4. *Lowering loop statements.*

Code example 3 displays three ways to implement semantically matching loops. Each approach includes an initializer, condition, iterator, and statement. Separate iterator statement shown in the examples is not required in `while` and `goto` statements but is included to visualize the transformation from a `for` loop. The initializer is executed once before entering the loop. This could include initializing local variables used by the loop. The condition is used to determine if the loop statement should be executed. When entering the loop, the statement is executed. After the statement has been executed, the iterator is executed. Finally, the loop returns to the condition. This is repeated while the condition holds. The `goto` loop displayed in Code example 3 could be converted into a `do` statement by removing the first goto statement because it would skip checking the condition before the first iteration.

```
// For Loop.
'for' '(' initializer ';' condition ';' iterator ')' statement
// While Loop.
initializer
'while' '(' condition ')'
'{'
    statement
    iterator
'}'
// Goto Loop.
initializer
'goto' goto_condition ';'

goto_body ':'
statement
iterator

goto_condition ':'
'if' '(' condition ')'
    'goto' goto_body ';'
```

Code example 3. *Lowering for loop.*

5.3 For Each

C# contains a `foreach` keyword that can be used to write a for-each statement. The statement is lowered during bytecode compilation because the CIL bytecode does contain a `foreach` statement. The for-each statement enumerates the elements of a collection, executing an embedded statement for each element of the collection (Statements - C# language specification, 2022). The collection is enumerated using an enumerator. An enumerator provides a way to iterate over all the elements stored in a collection while hiding the implementation details. The enumerator consists of two main things: a `Current` property and a `MoveNext()` method. The `MoveNext()` method advances the enumerator to the next element. The method returns a boolean value which tells if a move was made. After a successful move, the `Current` property can be used to access the current element. Before the first move and after the last element, the value of the property is undefined. The enumerator is retrieved from the collection using a `GetEnumerator()` method. The method returns an enumerator for the collection. This means that different collections can have different implementations for the enumerator. The `GetEnumerator()`, `Current`, and `MoveNext()` are matched during the bytecode compilation based on their signature. Signature-based matching allows the collection and enumerator to be implemented without any dependencies. Code example 4 displays a `foreach` statement before and after compilation. The example shows how the bytecode version is disassembled into a while loop.

```
// C#
Collection<int> collection = new Collection<int>();
foreach(int item in collection) {
    Console.WriteLine(item);
}
// C# -> CIL -> C#
Collection<int> collection = new Collection<int>();
Collection<int>.Enumerator enumerator =
   collection.GetEnumerator();
while(enumerator.GetNext()) {
    int item = enumerator.Current;
    Console.WriteLine(item);
}
```

Code example 4. *A for-each statement before and after compilation.*

In addition to signature-based matching during compilation, inheritance and interfaces can be used to implement and enforce the rules. The collections included in the `System.Collections` and `System.Collections.Generic` namespaces use various interfaces to enforce the signature-based rules. In addition to providing support for the `foreach` statements, the interfaces are used by the `System.Linq` namespace. The `System.Linq` namespace provides multiple methods for iterating and modifying collections using different interfaces.

5.4 Properties

In C# properties can be used to implement data encapsulation. Data encapsulation is a property of a well-written object (Bartoníček, 2014). Data encapsulation should be used to protect private data stored in objects. Commonly this is done by using accessors to access private members of a class. Properties provide a mechanism to write accessors. They are used similarly to fields. However, they consist of a get and set method. This allows them to be used as fields while maintaining the flexibility of accessor methods.

Properties are syntactic sugar which is compiled into a simplified bytecode. This includes converting the get and set methods into real methods and replacing the field uses with the methods. Some metadata about the property is stored in the bytecode. Properties are complicated since C# threats properties as fields and CIL as methods. The C# compiler must transform properties into methods during compilation. In some cases, the compiler may only have access to a compiled assembly, which means that the compiler must use the metadata stored in the bytecode to determine if a property is being used.

There are different types of properties. C# version 1.0 introduced properties that provided a field-like syntax to implement a get and set method. C# version 3.0 introduced auto-properties. Auto-properties provide a field and accessors without data validation. The bytecode versions of auto-properties include a field, a get method, a set method, and property metadata. C# version 6.0 extended auto-properties to support field initialization. Additionally, properties can exclude either method or change the visibility of one method. However, the changed visibility must be less than the property's visibility.

Code example 5 displays a simple property that is used to access a private field. It contains a get method that returns the value stored in the private field. It uses an arrow function which removes the need to use the return keyword. The property also contains a set method, which assigns a new value to the private field. The set method uses a value keyword, which represents the value given to the property. Code example 6 displays a compiled version of the property. Similar to the C# version, it contains a field and a property. However, the bytecode version includes two methods that the property references.

```
public class Example {
    private int m_value;
    public int Value {
        get => m_value;
        set => m_value = value;
    }
}
```

Code example 5. *C# class with property.*

When a CIL bytecode method is executed, a frame is added to the evaluation stack. Stack frames are used to limit the visibility of each method. This means that the evaluation stack is made out of frames, which contain the values used by methods. The stack frame provides each method with an empty evaluation stack, input array, and local variable array. Providing each method with an empty evaluation stack simplifies the overall process since methods can only work with their own evaluation stack. In summary, the process

virtual machine contains an evaluation stack that contains frames. Each frame contains an evaluation stack for instructions and values and an array for method parameters and local variables.

In Code example 6 the `get_Value()` method receives zero parameters, but the first instruction loads a parameter stored at the index zero. This is because instance methods receive the target object as an argument. The method works by first pushing the target object onto the evaluation stack. It then pushes an instruction to load a field from the target object at the top of the evaluation stack. The load operator consumes the target object from the stack and replaces it with the result. Finally, a return instruction will return the value stored in the evaluation stack. The `set_Value()` method in Code example 6 receives a parameter. However, it still receives the target object, which means that the input array contains two values. The first instruction pushes the target object onto the evaluation stack. The second instruction pushes the value parameter onto the evaluation stack. The third instruction consumes these values by storing the value in the object. Finally, the method returns.

```
.class public auto ansi beforefieldinit Example
    extends [System.Private.CoreLib]System.Object
{
    .field private int32 m_value

    .method public hidebysig specialname
        instance int32 get_Value () cil managed
    {
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: ldfld int32 Example::m_value
        IL_0006: ret
    }

    .method public hidebysig specialname
        instance void set_Value (
            int32 'value'
        ) cil managed
    {
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: ldarg.1
        IL_0002: stfld int32 Example::m_value
        IL_0007: ret
    }

    .property instance int32 Value()
    {
        .get instance int32 Example::get_Value()
        .set instance void Example::set_Value(int32)
    }
}
```

Code example 6. *Compiled C# class with property.*

Code example 7 contains an auto property version of the property displayed in Code example 5. Using an auto property removes the need to define a field to store the value. Additionally, the get and set methods are reduced to simple keywords. Auto properties do not allow method bodies because they are compiler generated. The bytecode version of auto property can be seen in Code example 8. Constructor and compiler-generated attributes have been excluded from the example. The compiled version of property in Code example 6 and auto property in Code example 8 provide matching bytecode with only a different field name. This means that a simple property and auto-property provide matching results at a bytecode level.

```
public class Example {
    public int Value { get; set; }
}
```

Code example 7. *C# class with auto property.*

```
.class public auto ansi beforefieldinit Example
    extends [System.Private.CoreLib]System.Object
{
    .field private int32 '<Value>k__BackingField'

    .method public hidebysig specialname
        instance int32 get_Value () cil managed
    {
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: ldfld int32 Example::'<Value>k__BackingField'
        IL_0006: ret
    }

    .method public hidebysig specialname
        instance void set_Value (
            int32 'value'
        ) cil managed
    {
        .maxstack 8
        IL_0000: ldarg.0
        IL_0001: ldarg.1
        IL_0002: stfld int32 Example::'<Value>k__BackingField'
        IL_0007: ret
    }

    .property instance int32 Value()
    {
        .get instance int32 Example::get_Value()
        .set instance void Example::set_Value(int32)
    }
}
```

Code example 8. *Compiled C# class with auto property.*

5.5 Indexers

In C# indexer allows an object to be indexed in the same way as an array (ECMA International, 2017, p. 322). Similarly to properties, indexers are syntactic sugar that is compiled into a simplified bytecode. Like properties, indexers are compiled into bytecode methods. Some key differences for indexers include no user-defined names, signature-based identification, always an instance member, and support for additional parameters (ECMA International, 2017, p. 323).

Code Example 9 displays how an indexer can be declared and used. Code Example 10 displays a shortened version of the compiled bytecode class. The example shows that the indexer has been compiled into separate get and set methods. Additionally, the indexer has been stored as property metadata in the bytecode.

```
public class Example {
    private int[] m_values;
    public int this[int index] {
        get => m_values[index];
        set => m_values[index] = value;
    }
}
// Using the indexer:
var example  = new Example();
example[0] = 0;
```

Code example 9. *C# class with indexer.*

```
.class public auto ansi beforefieldinit Example
    extends [System.Runtime]System.Object
{
    .field private int32[] m_values
    .method public hidebysig specialname
        instance int32 get_Item (
            int32 index
        ) cil managed
    // ...
    .method public hidebysig specialname
        instance void set_Item (
            int32 index,
            int32 'value'
        ) cil managed
    // ...
    .property instance int32 Item(
        int32 index
    )
    {
        .get instance int32 Example::get_Item(int32)
        .set instance void Example::set_Item(int32, int32)
    }
}
```

Code example 10. *Compiled C# class with indexer.*

6 BYTECODE INSPECTION

In this chapter, we will perform benchmarks and inspect the bytecode generated from different C# features. We will start by comparing the performance between an array and the generic `List<T>` included in the `System.Collections.Generic` namespace. Both the array and list can be used to store a sequence of elements. However, the list extends the functionality provided by an array. Comparing the performance between an array and a list should give us an idea of the possible performance impact the selected high-level data structure may have. After comparing arrays and lists, we will compare the performance between fields, properties, and indexers with bare-bone implementations to try and measure the performance impact of syntactic sugar.

Inspecting the C# version of the generic list shows that it uses an internal array to store the values. Access to these values is provided using methods, properties, and an indexer. When a list is created, it allocates an internal array with an initial capacity. Access to the internal array is limited based on a size variable. The size variable describes the number of elements stored in the list. New elements can be added using various add methods, increasing the size. Existing elements can be removed using various removal methods, decreasing the size. When values in the list are accessed, the size variable is used as the upper bound. The size is always smaller or equal to the capacity. By initially allocating a larger array than required, new elements can be added to the end without reallocating the entire array. The capacity is increased by reallocating the entire array into a new larger array. (List.cs, 2022)

Benchmarks are used to measure the performance of a given solution (Akinshin, 2019, p. 9). We are performing the benchmarks using the BenchmarkDotNet framework. Using a benchmarking framework helps with automating the process and avoiding different pitfalls. Arrays and lists were selected because they provide a flexible platform to perform benchmarks. They allow the benchmarks to be performed with various input sizes. Iterating over a sequence of elements requires the same action to be repeated multiple times. Accessing different elements prevents the runtime from optimizing the solution for a single input. Iterating over all the elements stored in an array requires a linear $O(N)$ loop. The linear time complexity means that the runtime should scale linearly with the input size. If a solution does not scale linearly, it may have some edge cases which make it unpredictable. In our case, the syntactic features may add unexpected overhead.

The benchmarked methods do not implement any platform-specific optimizations. Additionally, the methods do not contain any intentional heap memory allocations. Removing heap allocations removes the possibility of a garbage collection cycle, which could add additional overhead to the benchmark. In some cases, syntactic sugar in C# may produce hidden heap allocations by hiding the explicit object creation (Kokosa, 2018, pp. 481, 489–490). However, the hidden memory allocations in C# are always visible on

the bytecode level.

It is up to the virtual machine to optimize the code. The virtual machine defines automatic memory layouts for objects and inlines the code. Bound checking is one thing to consider when working with arrays. Bound checking ensures that a given index is within the array. Akinshinin (Akinshin, 2019, pp. 73–76) states that using arrays length in the condition of the loop can help the JIT compiler to eliminate bound checking. This is because it automatically uses the upper limit. It is also noted that these types of features should not be exploited while benchmarking.

The source code and used results are available on GitHub (Löppönen, 2022). All benchmarks were performed using a release build. The benchmarks are written for C# 9.0, compiled with Roslyn, and benchmarked using .NET 5.0. The benchmarks are divided into separate classes that contain a setup and the benchmark methods. The setup is used to create and populate the tested data structure. The data structures store 32-bit integers, starting from zero and increasing by one for each position. The 32-bit integer was selected for simplicity. The data structures use a generic type parameter to define the type of the stored objects. However, the benchmarking methods use an explicit type. The benchmarks were performed using three input sizes, including 1000, 10000, and 100000 elements. It should be noted that the largest input results in an overflow resulting in an incorrect result.

6.1 BenchmarkDotNet

BenchmarkDotNet is a .NET benchmarking framework. The framework can be used to transform methods into benchmarks. Transforming a method into a benchmark allows the framework to test its execution time and memory usage. In addition to benchmarking, the framework helps to avoid common benchmarking mistakes. The framework provides different attributes that are used to mark different test-related methods and fields. In addition to just testing methods, it provides support for setup and multiple input values. This provides a flexible framework for testing the performance with multiple inputs. (Akinshin, 2019, pp. 367–373; BenchmarkDotNet, 2021)

The framework supports multiple runtimes. It automatically generates isolated projects for each runtime setting and builds them in release mode. It then uses the input values and methods to generate all possible combinations. Each combination for a benchmark process is launched a few times to try and measure its performance. Each launch contains multiple iterations, where a single iteration includes invoking the benchmarked method. Before the actual benchmarking start, a few warmup methods are performed. A warmup run is used to verify that the JIT compiler has compiled the target method. An overhead warmup is used to evaluate the performance impact of the framework. A pilot run is used to determine how many iterations should be used to test the method. Finally, the benchmark iterations are executed, and a summary of the results is created. (How it works - BenchmarkDotNet, 2019)

6.2 Benchmark Methods

All the benchmarking methods iterate over the target data structure, counting the sum of all the stored elements. The tested data structures are allocated before the benchmark, resulting in zero heap memory allocation during the benchmarks. We are only testing the performance of accessing values stored in the targeted data structure. Only accessing values allows us to avoid reallocations resulting from modifying a targeted data structure.

There are a few things to consider when testing the performance. First, we consider where the value is stored. Based on the Common Type System, both the array and the list are stored on the heap. However, they can still be referenced in various ways. They can be stored in a field in an object, local variable, or received as an argument to a method. When a value is stored in a field, it requires few instructions to load when used inside a method. The method receives the owner object as the first argument. When a field is used, the owner object is pushed onto the evaluation stack, and then an `ldfld` instruction is used to load the value stored in the field. When a value is received as an argument or declared as a local variable, `ldloc` and `ldarg` instructions can be used to push the value onto the evaluation stack. Secondly, we can consider the non-indexed values used during the loop. For example, the length of a data structure is commonly used in the condition of the loop. However, if the length does not change, the length can be stored in a local variable. Storing the length in a local variable removes the need to load the length from the target object during each iteration. Finally, the implementation of a `foreach` loop is unknown to the user, which can yield widely different results.

There are multiple ways to implement linear loops. Because the C# compiler uses lowering to rewrite loops, we will only consider `for` and `foreach` loops. After lowering `for`, `while`, and `goto` statement loops produce matching bytecode. A `for` loop can iterate over the target data structure either forwards or backward. A forward loop starts from the first element and stops at the last element. A backward or a reverse loop starts from the last element and stops at the first element. The main difference between a forward and backward loop is hidden in the initializer and condition. Since the backward loops start from the last element, the length of the data structure is retrieved only once during the initialization. This allows the condition to compare a local value to a constant value.

Code example 11 displays a simplified class structure of a benchmark class. It shows a field used to store the benchmarked structure with two benchmark methods. The benchmark method is used to benchmark the performance when the target is stored in a field. The static benchmark method is used to benchmark the performance when the target is received as an argument. The static method still uses the same field as the regular method. However, the field is only loaded once before the static method is called. Additionally, Code example 11 shows the bytecode instruction used to access the target when received as a field or as an argument.

Code example 12 displays the bodies of different benchmark methods. The example displays how the target is iterated and how the value is accessed inside the loop. The

```
public class BenchmarkClass {
    private int[] m_array;

    [Benchmark]
    public int BenchmarkMethod() {
        var sum = 0;
        for(int i = 0; i < m_array.Length; i++) {
            //IL_0006: ldarg.0              // this
            //IL_0007: ldfld int32[]
            //     BenchmarkClass::m_array // this.m_array
            //IL_000c: ldloc.1              // i
            //IL_000d: ldelem.i4            // this.m_array[i]
            sum += m_array[i];
        }
        return sum;
    }

    [Benchmark]
    public int StaticBenchmarkMethod() {
        return StaticBenchmarkMethod(m_array);
    }

    private static int StaticBenchmarkMethod(int[] array) {
        var sum = 0;
        for(int i = 0; i < array.Length; i++) {
            //IL_0007: ldarg.0    // array
            //IL_0008: ldloc.1    // i
            //IL_0009: ldelem.i4  // array[i]
            sum += array[i];
        }
        return sum;
    }
}
```

Code example 11. *Benchmark class structure.*

method names have been shortened to single-character names shown and described in Table 2. These names will be used to display the results. Methods A to E are forward loops, and F and G are backward loops. Method A, a simple `for` loop, will always be used as the baseline performance measurement. Method B is a `foreach` loop. It will be excluded if the target does not implement the requirements. Methods C, D, and E are extensions to the `for` loop. They store the length, target, or both in a local variable before the loop starts. Method F is a backward loop, and Method G is a backward loop that stores the target in a local variable. The initializer in the backward loop retrieves the length of the array. Because the length is only retrieved once, it is not stored in a local variable separately.

Method	Description
A	For loop
B	Foreach loop
C	For loop, with a fixed length
D	For loop, with a local reference to the target
E	For loop, with a fixed length and local reference to the target
F	Reverse For loop
G	Reverse For loop, with a local reference to the target

Table 2. *Benchmark method descriptions.*

```
// Method A
for(int i = 0; i < m_array.Length; i++) {
    m_array[i];
}
// Method B
foreach(var element in m_array) {
    element;
}
// Method C
int length = m_array.Length;
for(int i = 0; i < length; i++) {
    m_array[i];
}
// Method D
int array = m_array;
for(int i = 0; i < array.Length; i++) {
    array[i];
}
// Method E
int length = m_array.Length;
int array = m_array;
for(int i = 0; i < length; i++) {
    array[i];
}
// Method F
for(int i = m_array.Length - 1; i >= 0; i--) {
    m_array[i];
}
// Method G
int array = m_array;
for(int i = array.Length - 1; i >= 0; i--) {
    array[i];
}
```

Code example 12. *Loops used to iterate over a collection of elements.*

Non-static Methods

	Array			List		
Method	Mean	StdDev	Ratio	Mean	StdDev	Ratio
A	658,8 ns	2,06 ns	1,00	1283,2 ns	7,73 ns	1,95
B	437,0 ns	1,47 ns	0,66	2706,7 ns	8,61 ns	4,11
C	651,8 ns	2,20 ns	0,99	722,0 ns	2,24 ns	1,10
D	670,8 ns	2,25 ns	1,02	694,1 ns	2,80 ns	1,05
E	672,7 ns	3,30 ns	1,02	693,4 ns	1,95 ns	1,05
F	653,1 ns	1,52 ns	0,99	724,3 ns	3,29 ns	1,10
G	658,9 ns	2,90 ns	1,00	694,2 ns	2,65 ns	1,05

Static Methods

	Array			List		
Method	Mean	StdDev	Ratio	Mean	StdDev	Ratio
A	668,7 ns	2,41 ns	1,02	692,9 ns	4,08 ns	1,05
B	446,2 ns	5,07 ns	0,68	2712,4 ns	17,92 ns	4,12
C	677,6 ns	3,66 ns	1,03	693,0 ns	2,86 ns	1,05
D	670,7 ns	2,56 ns	1,02	693,8 ns	2,95 ns	1,05
E	675,4 ns	2,05 ns	1,03	693,2 ns	2,91 ns	1,05
F	657,9 ns	2,47 ns	1,00	694,5 ns	2,95 ns	1,05
G	657,3 ns	2,87 ns	1,00	692,8 ns	2,52 ns	1,05

Table 3. *Array and List benchmark results (n = 1000).*

6.3 Baseline Performance

The baseline performance comparison includes performing all the previously mentioned benchmark methods for arrays and lists. Table 3 displays the benchmark results. The table displays the results for an input size of 1000. In total, the benchmarks were conducted using three different input sizes. The table is divided into two halves horizontally. The top half displays the performance of non-static methods, and the bottom half displays the performance of static methods. The mean column shows the mean runtime of the method. The StdDev column displays the standard deviation for the runtime. The ratio column displays the ratio $current\ mean/baseline\ mean$, where the baseline mean is the mean of the non-static method A for an array. The ratio column can be used as a guideline to find differences. However, it is only based on the mean and should not be used as a precise measurement.

Figures 5 and 6 visualize the results shown in Table 3. The figures display the mean and standard deviation of each method. The standard deviation is visualized by a vertical bar around the mean. Figure 5 displays all the methods. It can be seen that methods A and B contain some variation compared to the other methods. Figure 6 shows the same results

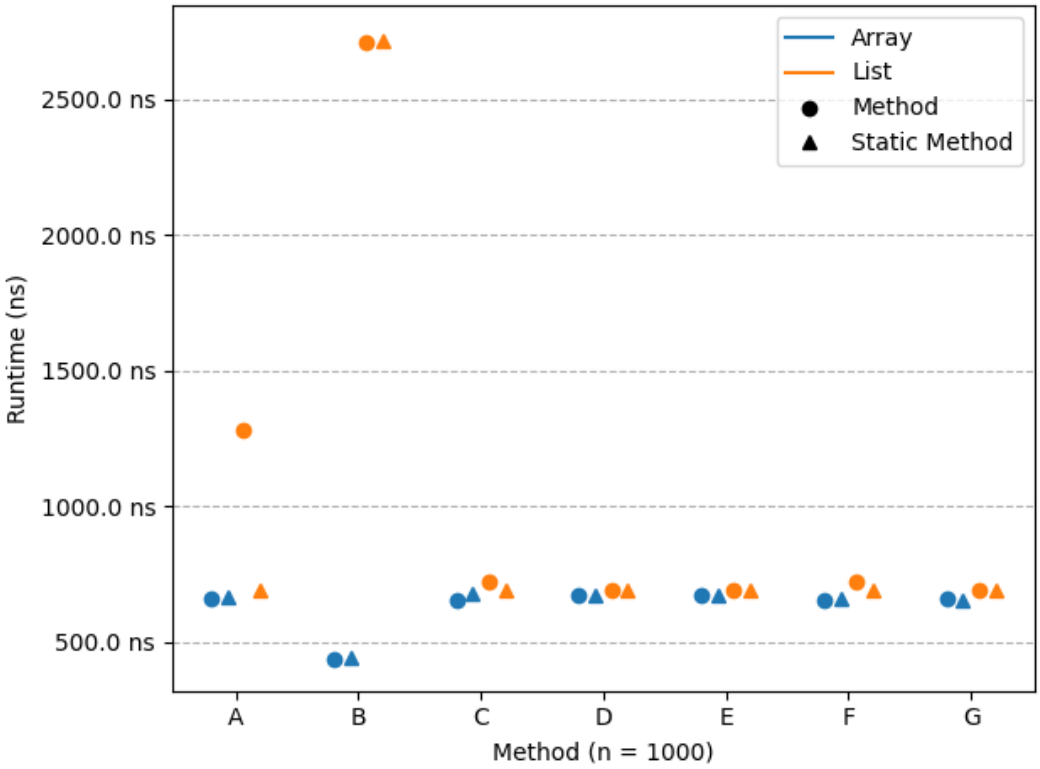

Figure 5. *Array and List benchmark results (n = 1000).*

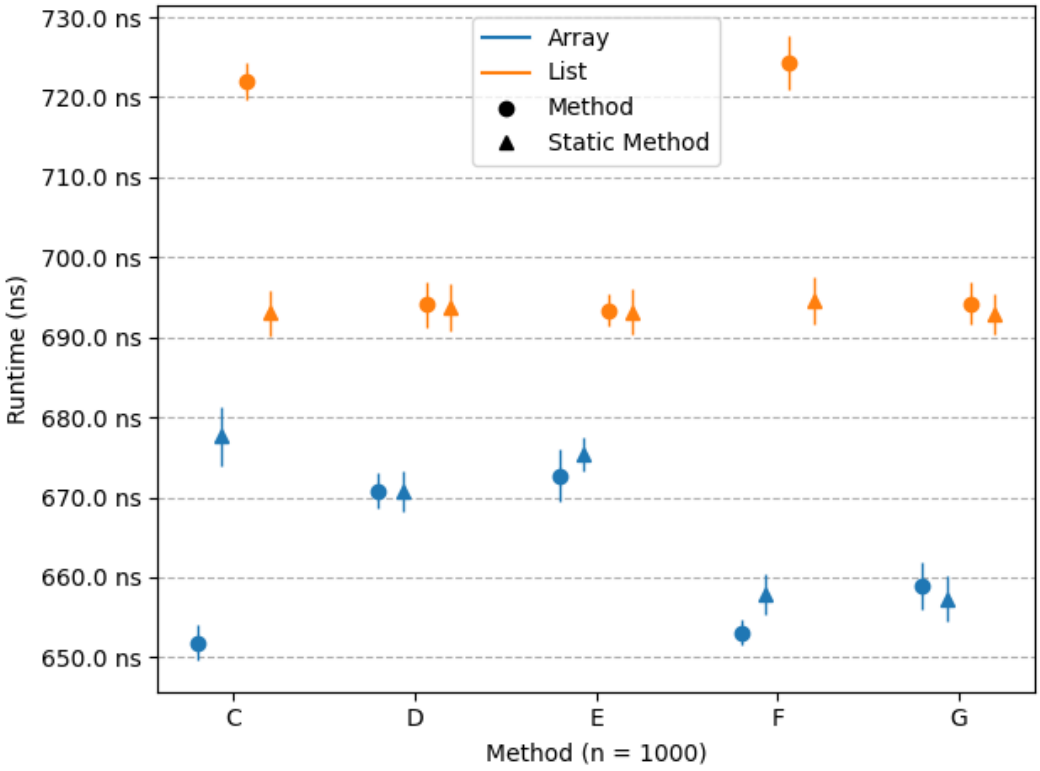

Figure 6. *Closeup of Array and List benchmark results for methods C-G (n = 1000).*

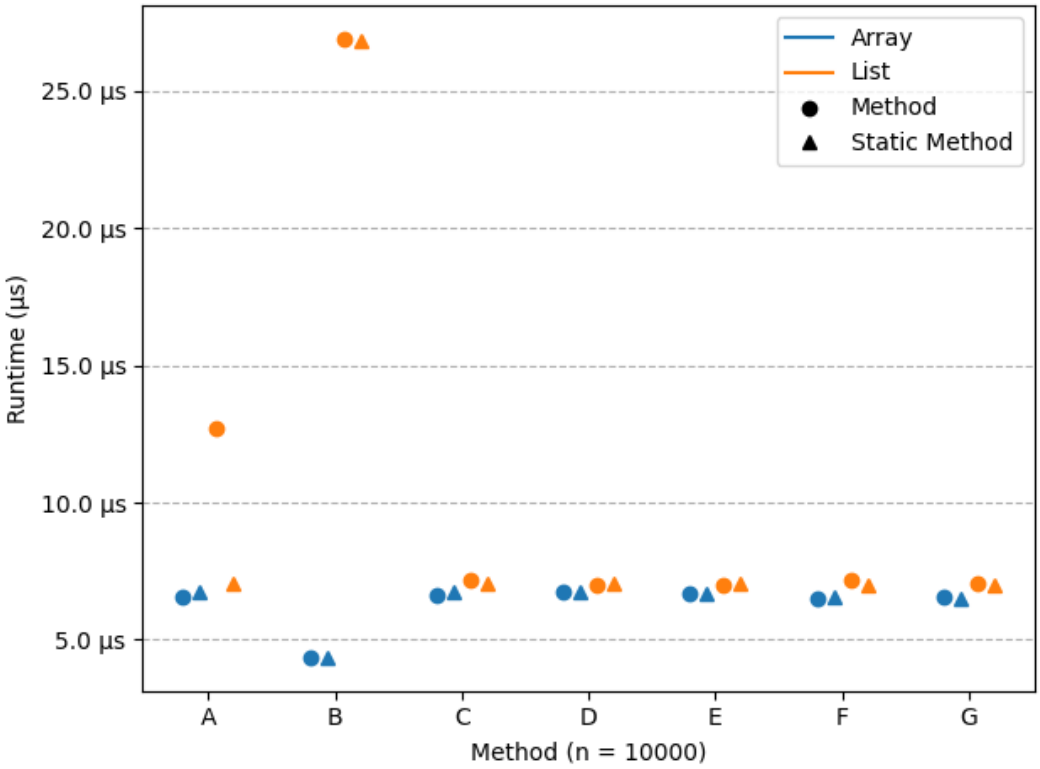

Figure 7. *Array and List benchmark results (n = 10.000).*

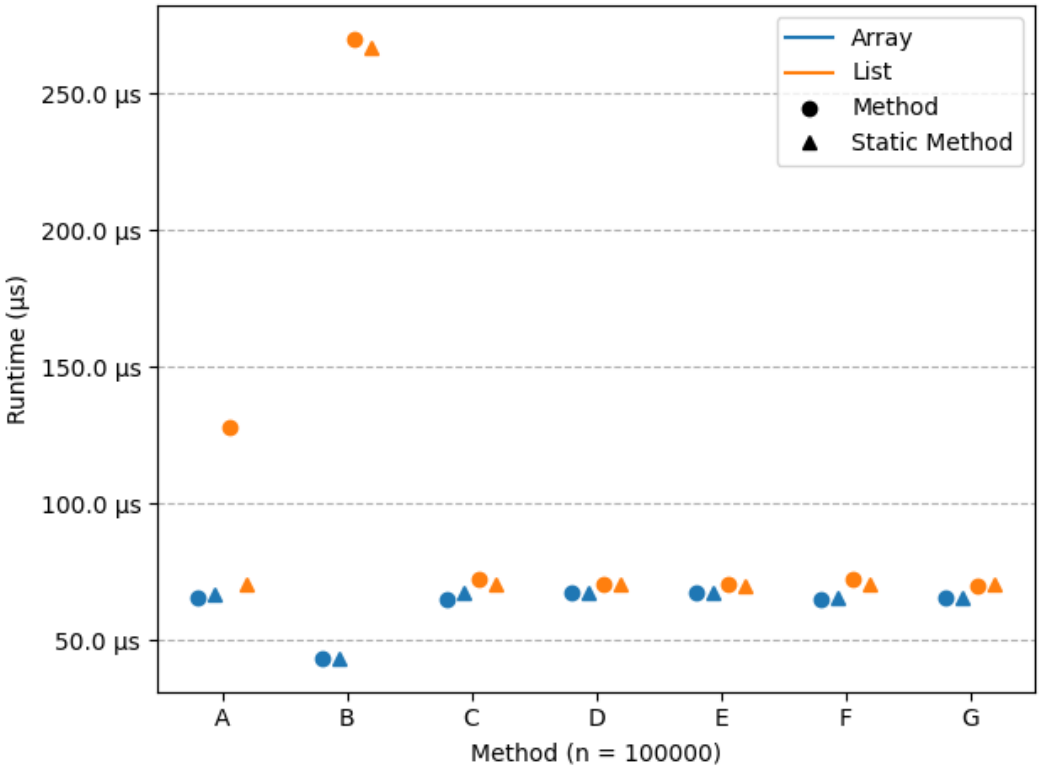

Figure 8. *Array and List benchmark results (n = 100.000).*

without A and B columns. These result show that the list adds additional overhead in all methods. Figures 7 and 8 display additional results for larger input sizes. They show that the runtime scales linearly with the input size as expected.

Looking at method A in Figure 5, we can see one outlier. The methods runtime almost doubles when the list is accessed through a field in the owner object. Method A produces almost identical bytecode for the non-static and static methods. The only difference is that the non-static version uses the `ldfld` instruction to load the list from the owner object. The load instruction could be considered as the reason for the longer runtime. However, the same changes can be seen when using an array, but the runtime does not change. This suggests that having to load an object from a field produces a slower runtime.

Looking at method B in Figure 5, we can see that there is a big difference between using an array and a list. It shows that enumerating over an array using the `foreach` statement provides the fastest runtime. Looking at the results produced by the list shows the slowest runtimes. However, it is important to note that the list provides a custom enumerator implementation. The implementation contains additional logic that is executed during the enumeration. At this point, the additional logic implemented by the list is unknown. This makes the performance comparison difficult and unfair since we do not know what the list is doing. We are not going to cover the additional logic added by the list. However, a quick overview shows that the list tracks its state and throws an error if the collection is modified during enumeration. This results in a safe way to enumerate over a collection.

Figure 5 shows that methods C to G provide similar results. These methods are better shown in Figure 6. Non-static methods C and F for a list show that storing the list's item count in a local variable can reduce the overall runtime. Storing the count in a local variable removes one `ldfld` instruction and almost halves the runtime compared to the regular `for` loop shown by method A. However, methods C and F containing the `ldfld` instruction and still results in a slower runtime than methods D, E, and G without the instruction. The generic list seems to provide a consistent performance when the list is stored in a local variable or received as an argument.

For arrays, method B seems to provide the best runtime. At first glance, methods B and D seem similar. However, they contain some minor details that can be used to explain the performance difference. Code examples 13 and 14 include small samples of methods B and D. Both samples display the beginnings of the methods and how the addition is performed. The beginnings display the code size, max stack size, and local variables. Code example 13 shows that method B has a max stack size of two and declares four local variables. Two of the local variables are compiler generated and do not have names. They have been named as "array" and "i" in the comments. Code example 14 show that method D results in a smaller code size with three local variables. However, it has a max stack size of three. Code example 13 shows that the `foreach` statement splits the addition into two parts. The first part includes loading the current value and storing it in a local variable. The second part includes loading the current sum and the current value and adding them together. The max stack size is reduced by using the local variables to store

the current value. In the Code example 14, the addition starts by pushing the current sum onto the evaluation stack, followed by loading the current value and adding them together. Having to load the current value on top of the current sum increases the stack size.

```
// Header size:  12
// Code size:  33 (0x21)
.maxstack 2
.locals init (
  [0] int32 sum,    //Total sum     "sum"
  [1] int32[],      //Target array  "array"
  [2] int32,        //Iterator      "i"
  [3] int32 v       //Current value "v"
)
// ...
// Counting sum:     Description         Stack
IL_000d: ldloc.1    //Push "array"        ["array"]
IL_000e: ldloc.2    //Push "i"            ["array", "i"]
IL_000f: ldelem.i4 //array[i]            [array[i]]
IL_0010: stloc.3    //"v" = array[i]      []

IL_0011: ldloc.0    //Push "sum"          ["sum"]
IL_0012: ldloc.3    //Push "v"            ["sum", "v"]
IL_0013: add        //"sum" + "v"         ["sum" + "v"]
IL_0014: stloc.0    //"sum" = "sum" + "v" []
// ...
```

Code example 13. *Sample of method B for an array.*

```
// Header size:  12
// Code size:  31 (0x1f)
.maxstack 3
.locals init (
  [0] int32 sum,    //Total sum    "sum"
  [1] int32[] arr, //Target array "array"
  [2] int32 i       //Iterator     "i"
)
// ...
// Counting sum:     Description           Stack
IL_000d: ldloc.0    //Push "sum"            ["sum"]
IL_000e: ldloc.1    //Push "array"          ["sum", "array"]
IL_000f: ldloc.2    //Push "i"              ["sum", "array", "i"]
IL_0010: ldelem.i4 //array[i]              ["sum", array[i]]
IL_0011: add        //"sum" + array[i]      ["sum" + array[i]]
IL_0012: stloc.0    //"sum"="sum" + array[i][]
// ...
```

Code example 14. *Sample of method D for an array.*

Non-static Methods

Method	Mean	Ratio	Code Size	Max Stack	Locals
A	658,8 ns	1,00	34	3	2
B	437,0 ns	0,66	33	2	4
C	651,8 ns	0,99	36	3	3
D	670,8 ns	1,02	31	3	3
E	672,7 ns	1,02	38	3	4
F	653,1 ns	0,99	36	3	2
G	658,9 ns	1,00	33	3	3

Static Methods

Method	Mean	Ratio	Code Size	Max Stack	Locals
A	668,7 ns	1,02	24	3	2
B	446,2 ns	0,68	28	2	4
C	677,6 ns	1,03	26	3	3
D	670,7 ns	1,02	26	3	3
E	675,4 ns	1,03	28	3	4
F	657,9 ns	1,00	26	3	2
G	657,3 ns	1,00	28	3	3

Table 4. *Array benchmark method bytecode statistics. Mean (n = 1000).*

Table 4 displays statistics about the compiled bytecode methods. It shows the runtime means and mean ratio compared to the non-static A method. The code size column displays the bytecode size. The max stack shows the max stack size stored in the bytecode generated by the compiler. Finally, the locals column displays the number of local variables declared by the method. In addition to the shown statistics, it should be noted that all methods receive a single argument. The non-static methods receive the owner object, and the static methods receive the array. There does not seem to be a direct relationship between the code size and execution time. Comparing matching static and non-static methods show that the code size decreases, but it does not increase the performance.

The baseline performance benchmark results have shown that arrays provide consistent runtimes when iterated. The main performance difference is achieved by using a `foreach` statement to enumerate the elements. The improved performance seems to be caused by minimizing the max stack size. The max stack size can be decreased by using local variables to store intermediate values rather than keeping them on the stack. The bytecode generated from the `foreach` statement for arrays can be replicated by a `for` loop by introducing a new local variable inside the loop to store the current value. A `for` loop that replicates the `foreach` loops bytecode for arrays can be seen in the Code example 15. The generic list included in the standard library seems to contain more variation. It seems to provide slower runtime when accessed through a field. The `foreach` statement provides the slowest runtime. However, the decrease in performance is caused

by additional checks made by the implementation of the list. The additional checks can be used to avoid errors caused by modifying the collection during enumeration. Because the list contains various implementation details, it is hard to tell to what extent syntactic sugar affects the performance.

```
private int[] m_array;
public int ReplicateForeach() {
    var sum = 0;
    var arr = m_array;
    for(int i = 0; i < arr.Length; i++) {
        var v = arr[i];
        sum += v;
    }
    return sum;
}
```

Code example 15. *For-statement that produces matching bytecode to the for-each statement when enumerating an array.*

The performance increase shown by the `foreach` loop seems to match Akinshin's (2019, pp. 407–411) Case study 2: Local variables, where the introduction of a local variable causes an increase in performance. In the case study, the addition of a local variable changed how a value is created. In our case, adding a local variable reduces the max stack size.

It should be noted that local variables behave differently in C# and CIL. In C#, local variables must be declared before being used, and they can have different scopes inside the method. In CIL, all the local variables are declared at the beginning of the method before any instructions. This means that the bytecode does not contain scopes for local variables and all the variable assignments use the memory reserved in the stack frame. Assigning a new value to a local variable allocated memory only if the statement on the right-hand side requires allocating memory. The difference is important as it can affect how the variable declaration is seen. In C#, a variable declaration inside a loop can be seen as a new variable on each iteration. However, in CIL, a single local variable is used during every iteration. CIL reusing a single local variable means that the C# variable declarations are mainly used for compile-time operations and checks.

6.4 Syntax Performance

In the previous section, we concluded that iterating arrays provides consistent performance. In most cases, the generic list included in the standard libraries provided similar performance to arrays. However, in some approaches, the generic list showed a decrease in performance. The list provides additional features using properties and indexers. However, these properties and indexer may contain some additional implementation details that were not covered. In this section, we aim to test the performance impact of properties and indexers. The performance impact is measured by providing access to an array using properties and indexers. By hiding an array behind syntactic sugar and performing the same benchmark methods, we can measure the performance impact of the syntactic sugar. Using bare-bone syntactic sugar to access the array allows us to test the performance cost added by the syntax sugar rather than any implementation details.

We aim to test the performance by storing an array inside a class and a struct. Using structs to implement data structures is not recommended, as they are meant for small and short-lived values (Choosing Between Class and Struct, 2021). The main difference between classes and structs is that classes are reference types, whereas structs are value types. Based on the Common Type System, this affects how they are treated and stored in the memory during execution. We test structs to see the performance impact of syntactic sugar. Properties and indexers do not have to be used to implement complex features. They can add simple functionality that utilizes existing values to generate new values. For example, they can return values based on other values stored in the object. Like the generic list, we use a generic type parameter to define the type of the stored elements. Using a generic type provides a flexible and reusable implementation of the tested method. Code example 16 demonstrates how a field, property, and indexer are declared using a generic type.

```
public class GenericClass<T> {
    // Array with generic type.
    public T[] m_array;
    // Auto-property array with a generic type.
    public T[] Array { get; set; }
    // Indexer with a generic type.
    public T this[int index] {
        get => m_array[index];
        set => m_array[index] = value;
    }
    // Read-only property for the arrays length.
    public int Length => m_array.Length;
}
```

Code example 16. *Using generic type to declare field, property, and indexer.*

Table 5 displays the benchmark results for classes, and Table 6 displays the benchmark results for structs. The tables have been divided into two horizontal halves. The top half shows the results for non-static methods, and the bottom half displays the results for static methods. The indexer implementation hides the array and does not implement a custom enumerator. Not having a custom enumerator prevents performing method B on indexers. The field column displays the benchmark results when storing the array in a public field. The property column displays the benchmark results when storing the array in an auto-property. The indexer column displays the benchmark results when storing the array in a private field and accessing the elements through an indexer. Storing the array in a private field prevents others from accessing its length. Additional read-only property is used to provide access to the length of the array.

Figure 9 visualizes the results for classes shown in Table 5 and Figure 10 visualizes the results for structs shown in Table 6. The figures display the mean and standard deviation of each method. The standard deviation is visualized by a vertical bar around the mean. Additionally, Figure 11 displays the results for properties and indexers in classes and structs. These are the same results shown in Tables 5 and 6.

Overall, the benchmark results show that the performance impact of syntactic sugar is minimal. However, in Figures 9 and 10 method A shows that using either property or indexer adds additional overhead in a specific condition. This condition requires a non-static method that stores the target object in a field. The decrease in performance is not seen in the static counterpart or when the target object is stored in a local variable. Additionally, Figure 11 shows that structs provide smaller overhead compared to classes.

Non-static Methods

	Field		Property		Indexer	
Method	Mean	StdDev	Mean	StdDev	Mean	StdDev
A	660,3 ns	3,98 ns	1303,1 ns	4,61 ns	1304,1 ns	5,08 ns
B	443,9 ns	1,36 ns	444,6 ns	1,45 ns	-	-
C	649,8 ns	2,12 ns	653,7 ns	3,52 ns	654,8 ns	2,75 ns
D	675,6 ns	2,53 ns	675,5 ns	3,18 ns	662,1 ns	2,22 ns
E	668,9 ns	2,79 ns	673,3 ns	2,44 ns	653,2 ns	2,74 ns
F	655,2 ns	4,42 ns	687,0 ns	22,47 ns	672,3 ns	11,51 ns
G	658,4 ns	2,41 ns	659,3 ns	2,02 ns	657,0 ns	3,88 ns

Static Methods

	Field		Property		Indexer	
Method	Mean	StdDev	Mean	StdDev	Mean	StdDev
A	661,5 ns	2,72 ns	658,9 ns	2,56 ns	659,8 ns	3,31 ns
B	556,5 ns	3,17 ns	435,2 ns	2,52 ns	-	-
C	651,8 ns	2,02 ns	652,5 ns	4,09 ns	655,2 ns	5,05 ns
D	673,6 ns	2,94 ns	675,0 ns	2,42 ns	662,3 ns	4,39 ns
E	674,5 ns	3,93 ns	675,9 ns	2,83 ns	652,2 ns	3,13 ns
F	654,7 ns	3,22 ns	652,7 ns	3,52 ns	655,9 ns	3,62 ns
G	658,8 ns	2,81 ns	658,1 ns	2,03 ns	653,5 ns	2,94 ns

Table 5. *Syntactic sugar performance on classes (n = 1000).*

Inspecting the bytecode for non-static A methods shows that the value in a given index is accessed in a different manner in each approach. All the approaches start by pushing the owner object onto the stack. The owner object is then used to load the field which contains the target object. When the target object is an array stored in a field, a `ldfld` instruction is used to load the array onto the stack. Once the array is on the stack, the current index is pushed onto the stack. Finally, a `ldelem.i4` instruction is used to access the element at the current index. Properties replace the array loading instruction with a method call that retrieves the array. Indexers push this further by implementing a get method that receives the current index as an argument and returns the element as a result. This results in the indexer hiding much of the implementation inside the get method. Additionally, classes and structs use different instructions to call the methods. Classes use a `callvirt` instruction while structs use `call` instruction. In addition to accessing the value at the current index, the length of the array is accessed in a similar manner.

Comparing non-static methods A, C, and F in Figure 11 show that storing the length in a local variable can remove the overhead seen in method A. Converting the condition of the loop into a comparison between local variables removes a load field instruction and

Non-static Methods

	Field		Property		Indexer	
Method	Mean	StdDev	Mean	StdDev	Mean	StdDev
A	660,2 ns	2,30 ns	916,0 ns	3,30 ns	917,4 ns	3,57 ns
B	432,0 ns	1,28 ns	443,1 ns	1,87 ns	-	-
C	655,7 ns	3,66 ns	650,7 ns	2,76 ns	652,5 ns	3,16 ns
D	674,7 ns	2,72 ns	673,3 ns	2,63 ns	674,8 ns	3,89 ns
E	675,6 ns	1,94 ns	678,0 ns	4,22 ns	673,5 ns	3,47 ns
F	658,4 ns	3,11 ns	653,2 ns	3,56 ns	651,6 ns	2,44 ns
G	657,4 ns	1,94 ns	658,1 ns	2,08 ns	657,1 ns	2,61 ns

Static Methods

	Field		Property		Indexer	
Method	Mean	StdDev	Mean	StdDev	Mean	StdDev
A	674,8 ns	2,71 ns	674,7 ns	4,64 ns	673,4 ns	2,82 ns
B	443,9 ns	2,22 ns	431,4 ns	0,78 ns	-	-
C	677,3 ns	1,99 ns	673,9 ns	2,91 ns	675,4 ns	3,20 ns
D	668,8 ns	2,70 ns	671,7 ns	1,59 ns	673,9 ns	2,89 ns
E	672,8 ns	2,06 ns	674,2 ns	2,87 ns	673,6 ns	2,59 ns
F	656,4 ns	2,67 ns	666,5 ns	8,70 ns	657,9 ns	2,67 ns
G	657,8 ns	2,68 ns	657,3 ns	2,32 ns	657,5 ns	2,19 ns

Table 6. *Syntactic sugar performance on structs (n = 1000).*

a method call from each iteration.

Static methods seem to provide consistent performance. This can be seen in Figure 11. The static methods do not include the initial loading of the target field. The static methods receive the target object as an argument. Receiving the target object as an argument allows the argument to be loaded onto the stack. Once the target object is on the stack, the required method can be called. The main difference between the static methods is the instruction used to load the arguments. Arguments with a reference type are loaded onto the stack with a `ldarg` instruction. Arguments with a value type are loaded onto the stack with a `ldarga.s` instruction.

Overall, the non-static method A seems to provide a special condition that adds additional overhead to the performance. The additional overhead can be removed by moving a `ldfld` instruction outside the loop. The instruction can be moved out of the loop by storing the target object in a local variable or by receiving it as an argument. Otherwise, using properties or indexers does not seem to add significant overhead to the performance.

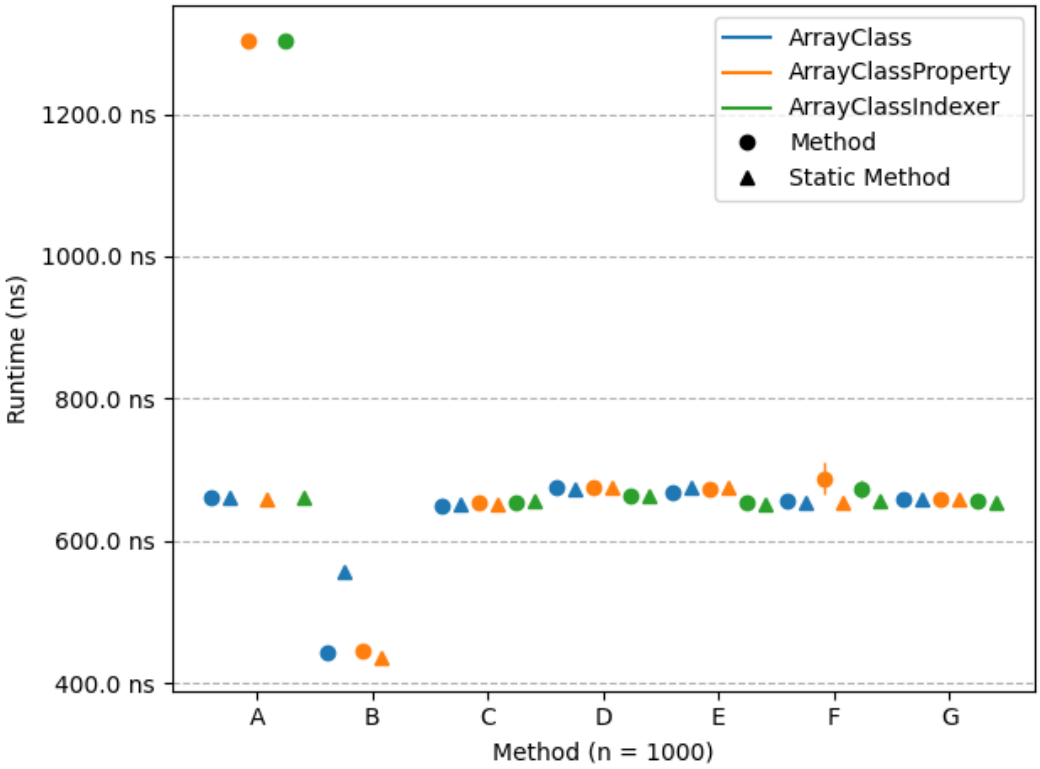

Figure 9. *Syntactic sugar performance on classes (n = 1000).*

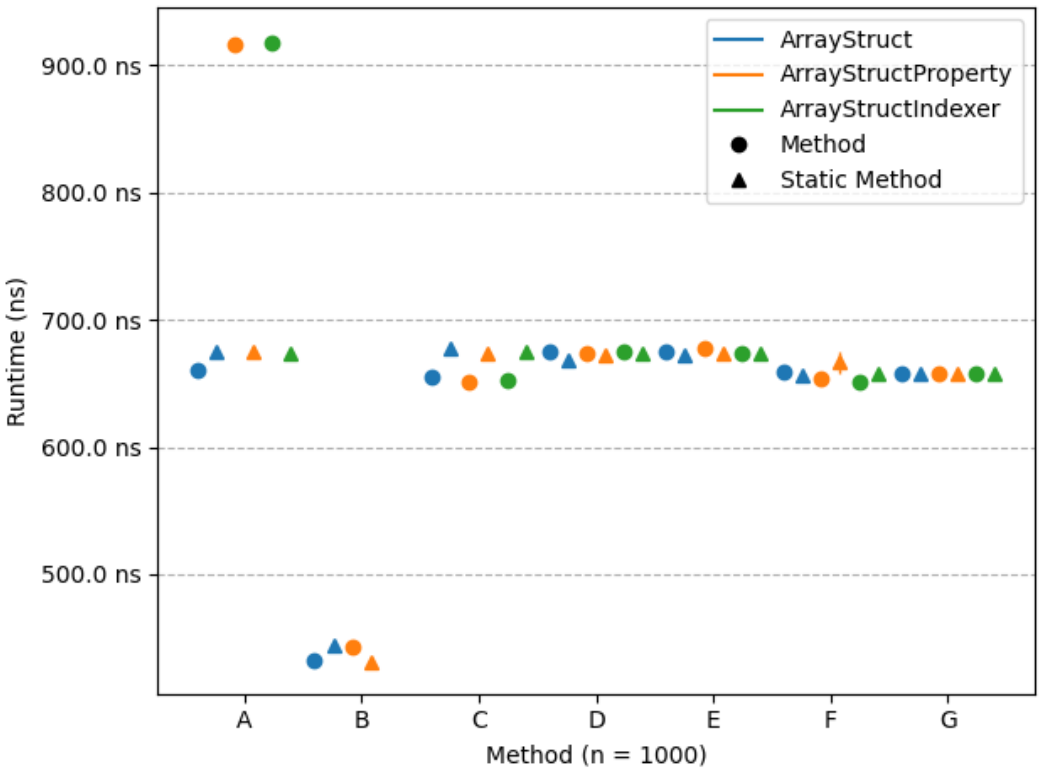

Figure 10. *Syntactic sugar performance on structs (n = 1000).*

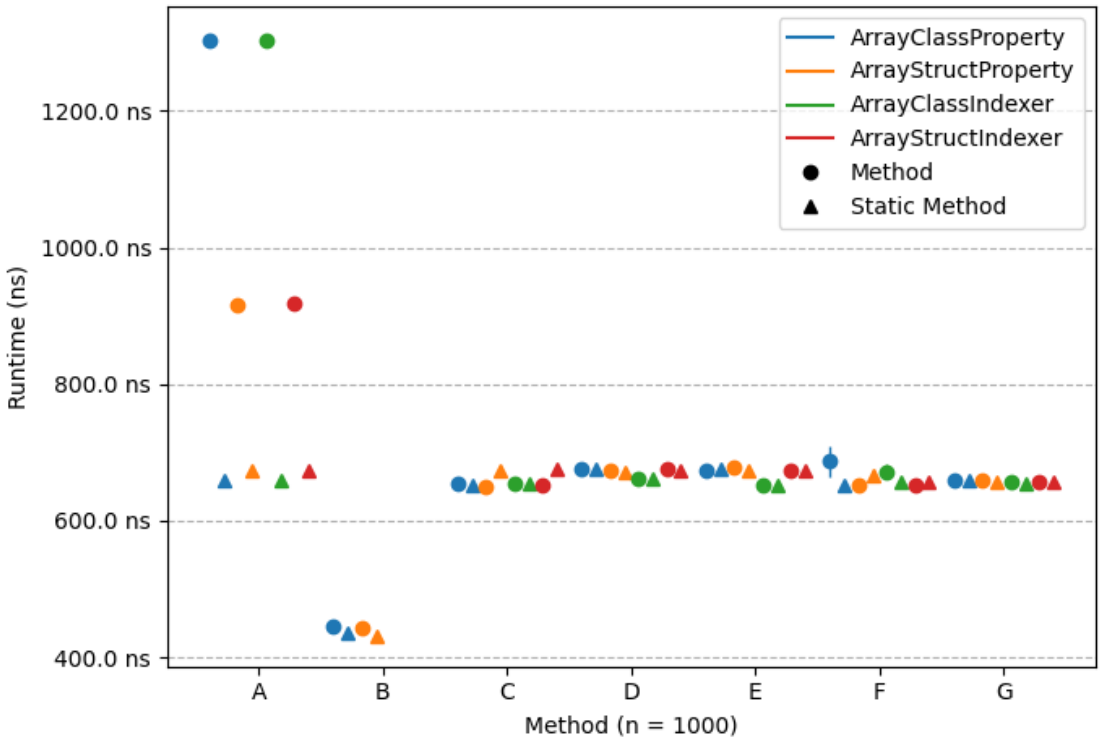

Figure 11. *Properties and indexers on classes and structs (n = 1000).*

6.5 Discussion

We have shown how various C# features get transformed during bytecode compilation. We have also performed benchmarks on the high-level changes. We chose to benchmark only two loop statements because multiple loop statements produce matching bytecode after being lowered during compilation. Lowering is used to rewrite the loops with simplified constructs, resulting in a matching bytecode. We also noted that there are some differences in local variables between C# and CIL. In C#, local variables can be declared when needed. The declaration also affects the scope of the variable. In CIL, all the variables are declared before any instructions, and they do not have scopes.

The benchmark results showed that arrays and lists generally provide similar runtimes. Arrays provide consistent performance, with room for small improvements. The best performance for arrays is provided by the `foreach` statement, which uses local variables to store the target object and the enumerated value. Storing the enumerated value in a local variable removes the need to hold it on the stack. The generic list showed more variation in the results. In some cases, the generic list added additional overhead to the performance. When the list is used through a field, it adds additional overhead to the performance. The overhead can be removed by storing the list in a local variable before the loop or by receiving the list as an argument. These changes simplify accessing the list inside the loop. The `foreach` statement provided the slowest runtime for the generic list. The list contains a custom enumerator used during the `foreach` statement. The custom enumerator contains implementation details that were not covered. Having

custom implementation details makes the performance comparison harder since the compared implementations may provide different features.

Simple properties and indexers without additional code do not add significant overhead to the performance. Additionally, the runtime for all features scaled linearly to the input size as expected. Properties and indexers seem to decrease the performance when used in a method through a field. In addition to classes, the same decrease in performance can be seen when using structs. This performance decrease is similar to the one seen in the generic list. The performance decrease can be avoided by storing the targeted data structure in a local variable before the loop and using the local variable inside the loop. Storing the targeted data structure in a local variable reduces the number of instructions required to access the values. Using an array stored in a field did not show the same performance decrease. The main difference between using an array and syntactic sugar to access an array is the replacement of direct bytecode instructions with method calls.

The code size of the benchmarked methods did not seem to affect the performance. However, the benchmarked methods were under 40 bytes long. In addition to the code size, the local variable count did not seem to affect the performance. All methods received one argument and had two to four local variables. A decrease in the max stack size seemed to increase the performance. However, a decrease in the max stack size was seen only in one method.

Our benchmarks focused on the runtime performance of the tested features. Additional benchmarks could be performed on the startup cost of generics, properties, and indexers. These benchmarks could be used to determine how much these features affect the duration of the initial JIT compilation. The duration of the initial JIT compilation may impact the overall performance of a short-running process. The bytecode inspection could be expanded to include the JIT-compiled instructions. The JIT-compiled instructions would be platform-specific instructions that could include platform-specific optimizations. However, they could provide better insight in to the performance.

7 CONCLUSION

Process virtual machines provide a platform-independent environment for running applications. They are executed inside an operating system just like any other process. Process virtual machines provide platform independence by translating platform-independent bytecode into platform-specific machine code. Using bytecode to deliver the application allows the same code to be deployed on multiple platforms. Additionally, a bytecode language can enable cross-language integration by using metadata to define shared properties. This allows multiple high-level languages to target a single bytecode language. By combining metadata with the bytecode instructions, the bytecode can form self-describing software components.

Compilers are used to translate one language to another. In this thesis, high-level code was compiled into bytecode. During the compilation, the high-level code was transformed into stack-based bytecode. Additionally, the compilation process included the removal of syntactic sugar. Lowering is used to rewrite complex semantics using simpler ones. Lowering is also used to simplify loop statements. Stack-based bytecode includes pushing values onto an evaluation stack. When a value is used, it must be on top of the stack. Once used, the value is unavailable for subsequent uses. Multiple uses require multiple copies.

The .NET virtual machine contains an execution engine to execute bytecode and a garbage collector to manage memory. The execution engine uses a just-in-time compiler named RyuJIT to translate bytecode instruction into machine code. RuyJIT is a single-tier JIT compiler, which means that a method is JIT compiled during the first call, and the resulting machine code is stored and used for subsequent calls. The garbage collector is a generational garbage collector, which divides the objects into generations. By dividing the objects into generations, the garbage collector can focus on a single generation that is likely to contain dead objects.

Our benchmarks focused on comparing arrays, lists, properties, and indexers. The baseline benchmarks showed that the `foreach` statement provides the fastest way to iterate over an array. The performance increase seems to be caused by the statement producing a smaller maximum stack size compared to a `for` statement. However, the resulting bytecode can be replicated using a `for` statement, by storing the current value in a local variable at the beginning of the loop statement. Overall, arrays and lists provided similar results, with the list providing a small overhead in most cases and noticeable overhead in a few cases. The list added overhead when used inside a non-static method through a field and when enumerated using the `foreach` statement. The for-each statement provided overhead to the method because it provides a custom implementation for enumerating the list. The custom implementation makes the performance comparison harder because it provides different features. Additional benchmarks were performed for properties and indexers to see the cost of syntactic sugar without the implementation de-

tails included in the generic list. The additional benchmark showed the same overhead when the target object was used inside a non-static method through a field. Additionally, the overhead was smaller when using structs compared to classes. The overhead could be removed by storing the target object in a local variable before the loop. In all other cases, properties and indexers did not add noticeable overhead. It seems that local variables can be used to reduce the amount of instruction required to access values.

Overall, syntactic sugar does not seem to have a noticeable impact on performance. The main difference is that direct bytecode instructions are replaced with method calls. In the cases where syntactic sugar caused a decrease in performance, it could be easily removed by storing the targeted data structure in a local variable before the loop. Storing the targeted data structure in a local variable reduced the number of instructions required to access the values. Additionally, storing the current value in a local variable inside the loop before using it could be used to decrease the maximum stack size required by the method. Reducing the maximum stack size seems to increase the performance. Based on our benchmarks, the introduction of new local variables in C# can be used to improve performance in certain cases.

REFERENCES

.NET Programming Languages. (2022). Retrieved January 28, 2022, from https://dotnet.microsoft.com/en-us/languages

.NET Standard. (2021). Retrieved November 18, 2021, from https://docs.microsoft.com/en-us/dotnet/standard/net-standard

Akinshin, A. (2019). *Pro .NET Benchmarking* (1st ed.). Apress. https://doi.org/10.1007/978-1-4842-4941-3

Bartoníček, J. (2014). Programming Language Paradigms & The Main Principles of Object-Oriented Programming. *Bulletin (Prague College Centre for Research & Interdisciplinary Studies)*, *2014*(1), 93–99.

BenchmarkDotNet. (2021). Retrieved January 19, 2022, from https://benchmarkdotnet.org

Brosgol, B. M. (2010). A comparison of generic template support: Ada, C++, C#, and Java. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6106*, 222–237.

Choosing Between Class and Struct. (2021). Retrieved January 22, 2022, from https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct

Companies using Mono. (2020). Retrieved January 10, 2022, from https://www.mono-project.com/docs/about-mono/showcase/companies-using-mono/

Costa, R., & Rohou, E. (2005). Comparing the size of .NET applications with native code. *Proceedings of the 3rd IEEE/ACM/IFIP international conference on hardware/-software codesign and system synthesis*, 99–104.

ECMA International. (2012). *Standard ECMA-335 - Common Language Infrastructure (CLI)* (6th ed.). http://www.ecma-international.org/publications/standards/Ecma-335.htm

ECMA International. (2017). *Standard ECMA-334 - C# Language Specification* (5th ed.). http://www.ecma-international.org/publications/standards/Ecma-335.htm

Eimouri, T., Kent, K. B., Micic, A., & Taylor, K. (2016). Using Field Access Frequency to Optimize Layout of Objects in the JVM. *Proceedings of the ACM Symposium on Applied Computing*, *4-08-*, 1815–1818.

Forstall, B. (2018). *The RyuJIT transition is complete!* Retrieved February 2, 2022, from https://devblogs.microsoft.com/dotnet/the-ryujit-transition-is-complete/

Fruja, N. G. (2006). Type Safety of Generics for the .NET Common Language Runtime. *Programming Languages and Systems*, 325–341.

Fruja, N. G. (2008). Towards proving type safety of .NET CIL. *Science of Computer Programming*, *72*(3), 176–219. https://doi.org/https://doi.org/10.1016/j.scico.2008.05.004

How it works - BenchmarkDotNet. (2019). Retrieved January 28, 2022, from https://benchmarkdotnet.org/articles/guides/how-it-works.html

Kokosa, K. (2018). *Pro .NET Memory Management For Better Code, Performance, and Scalability* (1st ed. 2018.). Apress.

List.cs. (2022). Retrieved March 1, 2022, from https://source.dot.net/#System.Private.CoreLib/List.cs

Löppönen, T. (2022). *DotNetBytecodeInspection*. https://github.com/TapioLopponen/DotNetBytecodeInspection

Lowering - dotnet/roslyn. (2022). Retrieved January 13, 2022, from https://github.com/dotnet/roslyn/tree/main/src/Compilers/CSharp/Portable/Lowering

Maierhofer, M., & Ertl, M. A. (1997). Optimizing stack code. *Forth-Tagung, Ludwigshafen*.

Managed Execution Process. (2021). Retrieved February 8, 2022, from https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process

Meijer, E., & Gough, J. (2000). Technical Overview of the Common Language Runtime.

Mono. (2022). Retrieved January 10, 2022, from https://www.mono-project.com/

Mono Application Portability. (2020). Retrieved January 10, 2022, from https://www.mono-project.com/docs/getting-started/application-portability/

.NET Team. (2013). *RyuJIT: The next-generation JIT compiler for .NET*. Retrieved February 2, 2022, from https://devblogs.microsoft.com/dotnet/ryujit-the-next-generation-jit-compiler-for-net/

Park, J., Park, J., Song, W., Yoon, S., Burgstaller, B., & Scholz, B. (2011). Treegraph-based Instruction Scheduling for Stack-based Virtual Machines. *Electr. Notes Theor. Comput. Sci.*, *279*, 33–45. https://doi.org/10.1016/j.entcs.2011.11.004

Parnin, C., Bird, C., & Murphy-Hill, E. (2012). Adoption and use of Java generics. *Empirical Software Engineering: An International Journal*, *18*(6), 1047–1089.

RyuJIT Tutorial. (2021). Retrieved February 2, 2022, from https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-tutorial.md

Sebesta, R. W. (2012). *Concepts of Programming Languages* (10th ed., international ed.). Pearson/Education.

Shi, Y., Casey, K., Ertl, M., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, *4*(4), 1–36.

Statements - C# language specification. (2022). Retrieved January 9, 2022, from https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements

Sudha, M., Harish, G. M., Nandan, A., & Usha, J. (2013). Performance Analysis of Kernel-Based Virtual Machine. *International Journal of Computer Science & Information Technology*, *5*(1), 137–144.

Syntactic Sugar - Wikipedia. (2021). Retrieved January 13, 2022, from https://en.wikipedia.org/wiki/Syntactic_sugar

Urma, R.-G. (2014). *Alternative Languages for the JVM*. Retrieved January 28, 2022, from https://www.oracle.com/technical-resources/articles/java/architect-languages.html

Warren, M. (2017). *Lowering in the C# Compiler*. Retrieved January 13, 2022, from https://mattwarren.org/2017/05/25/Lowering-in-the-C-Compiler/

What's new in .NET 5. (2022). Retrieved March 6, 2022, from https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-5